IMPLEMENTATION OF AN INTERFACE PROCESSOR AND DESIGN OF ALGORITHMS FOR 'SASP'

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

by
SAMIT CHAUDHURI

to the
DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
APRIL, 1989

EE- 1989-M-CHA-IMP

CERTIFICATE

This is to certify that the work embodied in this thesis titled, '**Implementation of an Interface Processor and Design of Algorithms for SASP**', has been carried out by Mr. Samit Chaudhuri under my supervision and same has not been submitted elsewhere for a degree.

Amit Mahanta

30/5/8

(Dr. A.Mahanta)

Assistant Professor

Department of Electrical Engg.

Indian Institute of Technology

Kanpur

ABSTRACT

A linear systolic array (SASP— Systolic Array Signal Processor) for DSP applications has been proposed. SASP is divided in two major functional blocks - interface unit, and processor array. The bulk of the computing power comes from the processor array which consists of a linear array of locally interconnected identical processing elements called 'cells'. Data for computation is dumped onto the interface unit by the external host. During execution, the interface 'pumps' data into the processor array in the required regular pattern and generates addresses used by the cells to access their local data memories. Each individual cell can be microprogrammed through the interface which itself also works under microcode control. A microengine and an address generation unit of the interface have been implemented and a simple cell has been made to execute some common DSP algorithms. The mapping of algorithms on this linear array structure is discussed.

Acknowledgements

It has been a great pleasure to write a thesis in the stimulating environment of the Dept. of Electrical Engineering, IIT, Kanpur. A dedicated community of fellow students and the ever helpful faculty members had helped to bolster the nerves at those trying moments when exhaustion and fatigue threatened to jade the working spirit. I am indeed happy to thank all my friends and teachers at the ACES.

Any amount of thanks is dwarfed before the constant encouragement and attention I got from my advisor, Dr. A. Mahanta. His friendly countenance is a precious gift for any student.

I am especially grateful to Dr. M.U. Siddiqi for allowing me to use the Image Processing Lab facility and for providing me with a project fellowship during the last few months of my graduate programme. Dr. S.K. Mullik's gracious permission for using the ACES 205 Laboratory proved to be a boon.

I profited from the discussions with Sanjeev Khadilkar, whose sincere advice was of the rarest kind. I am lucky to have friends like Sudip, Majee, Sandhu, Dada, Kushal, Venkatesh, Dipankar-da, Subrata-da, Ashoke, Santanu, Snehangshu-da. Leisurly chats and timely help from this friendly fraternity will adorn my memories of the short stay at IIT, Kanpur. The cheerful company of D.S.Banerjee kept me sane after the tiring work-sessions with Usman, who was always eager to cooperate. I am grateful to these two friends for an ideal blend of, on-the-work and off-the-work environments.

CONTENTS

1. Introduction 1

1.1 Computational Requirements in Digital Signal Processing and Parallel Architectures 1

1.2 Array Processor Architectures 3

1.3 Systolic Arrays 7

1.4 Objectives and Scope of the Work 8

2. The SASP Architecture 10

2.1 Introduction 10

2.2 System Architecture 11

2.3 States of Operation of the 'SASP' 12

2.4 Intercell Communication 13

2.5 Signals on Broadcast Bus 14

2.6 Features of SASP Architecture 16

3. Mapping Algorithms onto Systolic Arrays 17

3.1 Introduction 17

3.2 Descriptions of Algorithms 18

3.3 Dependence Graph 20

3.4 Mapping Methodology 20

3.4.1 Design step 1. Mapping Algorithm onto DG 20

3.4.2 Design step 2. Mapping DG onto SFG 21

3.4.3 Design step 3. Mapping SFG onto Systolic Array 27

3.5 Large Size Problems 29

4. Matrix Computations on Systolic Array 35

4.1 Introduction 35

4.2 Mapping of LU-decomposition 36

4.3 Mapping of QR-decomposition 45

4.4 Eigen Values of a Symmetric Matrix 50

5. **Implementation of Interface Controller** 55

5.1 Introduction 55

5.2 Functions of the Interface 56

5.3 Block Diagram Description of the Interface 57

5.4 Designing 'IFCU' and 'AGU' 61

5.4.1 Interface Control Unit (IFCU) 61

5.4.2 Address generation unit (AGU) 64

5.4.3 Downloading of Microcode 64

5.4.4 Data Transfer Between 'IFCU' and 'AGU' 66

5.5 Circuit Description 69

5.5.1 Microcode Loading 71

5.5.2 Execution of Microcode 73

5.5.3 Use of Microcode Bits 74

6. **Cell for 'SASP'** 77

6.1 Introduction 77

6.2 Cell for 'SASP' 78

6.3 Design of a Simplified Cell 81

6.3.1 Block Diagram Description 82

6.3.2 Hardware Description 84

6.4 Execution of Algorithms on the Cell 88

6.5 A Software Utility to Use 'SASP' 90

7. **Summary and conclusions** 95

Chapter 1

INTRODUCTION

§ 1.1 *COMPUTATIONAL REQUIREMENTS IN DIGITAL SIGNAL PROCESSING & CURRENT PARALLEL ARCHITECTURES*

Very high speed computation is required in a variety of digital signal processing applications. The increasing utilization of TV imaging in geophysical, medical and industrial environments has led to an increasing demand for high speed signal processing. These applications require processing in 'real time' or 'near real time ' environments, where the data rate of processed images are almost the same as that of input images. For example, an image size of 512X512 pixels and a frame rate of one frame per millisecond gives an input data rate of 262 Mpixels/sec. In a vision processing system, the job of recognizing an object and checking its geometric and physical properties would require an approximate processing speed of

10ops/pixelX512X512X1000 = 2621 Mops (Million operations per second) [SYKung88].

Huge amount of computing power is required to support such processing demands. But present day technology has almost reached the stage where

significant speed-up can no more be squeezed out of a single processor. Therefore, fast single processor sequential computers are gradually giving way to concurrent computing systems which try to exploit the parallelism, simultaneity and pipelinability of the events occurring in a computing process [Laz86].

Modern parallel computers can be classified in three architectural categories: pipeline processors, multiprocessor systems, and array processors [Hwang84]. Structures of the first kind exploit temporal parallelism while the second one achieves asynchronous parallelism through a set of interactive processors. These two classes of architecture are mainly found in general purpose computer domain. The third type of structures find applications as special purpose computers intended for applications like signal and image processing.

Although the general purpose supercomputers well live up to their commitment of high speed computation (e.g. 100 Mop vector operation in Cray-1; Cray-2 claims an 16fold increase in performance; 10 Gop peak performance in ETA-10 etc.), their immense 'number crunching' power may appear elusive in a particular area of operation. The reasons can be manifold – general purpose computers can not fully exploit the parallelism of all classes of problems [Dew84] ; moreover, I/O bandwidth requirements may stand in the way to improved performance. Besides this, the cost of a general purpose supercomputer may not justify its use in a narrow, specific field of application.

Thus, before building a special purpose machine, there should be an attempt to meet two major justifications[Stone85]:

1) The special purpose system performs a function faster than any existing general purpose computer.

2) The cost of developing the special purpose system should be justifiable in spite of its narrow range of application.

The first justification is found in signal processing applications (e.g. pattern recognition, speech, sonar, radar, seismic, weather, astronomical, medical signal processing), where large number of computations must run in real time to support enormous throughput rates. As regards the second justification, recent advances in VLSI technology have made it possible to construct large, special purpose systems at relatively low cost, since such systems can be composed of collections of powerful, general purpose VLSI parts. Extra speed-ups in execution for a specific class of problems can be achieved by relating the algorithm and architecture more closely and thereby exploiting the characteristics of the problem solving method.

§ 1.2 *ARRAY PROCESSOR ARCHITECTURES*

Array processor architectures are extensively used for special purpose computing structures. An array processor uses multiple synchronized processing elements (PE's) which operate in a spatially parallel manner. The difference between array processors and multiprocessor systems is that the PE's in an array processor operate synchronously, but processors in a multiprocessor system may operate in asynchronous manner. In this section, different array processor architectures will be discussed with a brief review of some existing architectures.

i) **Single Instruction Multiple Data (SIMD) arrays**

An SIMD array is a synchronous array of processing elements (PE's) with local memories and local connectivity between them. All the PE's operate under control of a central control unit (CU). Instructions are broadcast to the PE's which execute the same instructions on different data sets. Examples of SIMD systems are ILLIAC IV, Massively Parallel Processor (MPP), Distributed Array Processor (DAP) etc.

ii) **Multiple Instruction Multiple Data (MIMD) arrays**

An MIMD system consists of a number of PE's, each of which has its own data, control unit and program. Thus, the overall processing task can be distributed over different PE's. MIMD machines are prone to communication bottleneck and this factor has affected their popularity in spite of their highly flexible structure. They are more suitable to handle irregular algorithms. The dataflow machine is an MIMD computer in which an instruction is ready for execution as soon as its operands become available.

A combination of SIMD and MIMD principles is also used in order to have a balance between simple communication and architectural flexibility. Examples of such systems are PASM, pyramid architectures etc.

• MPP

It is a specialized system whose design was motivated by needs to process millions of satellite image data quickly. MPP employs a 128X128 array of PE's using SIMD mode of parallelism. The data stream is processed by the array unit (ARU), while the instructions are processed by the array control unit (ACU). The staging memory acts as a buffered I/O path of the ARU and allows transfer of data with the outside world in the optimum format. ACU is divided into three

parts which allows array arithmetic, scalar arithmetic, and input-output operations to take place in parallel [BAT85]. MPP is capable of 32 bit 470 MFlops ALU operations (i.e. addition, subtraction & logical operations) and 32 bit 291 MFlops multiply operations.

* PASM

This system is designed mainly to be a research tool for studying the use of large scale parallelism in problem domain of image understanding. It is a partitionable SIMD/MIMD system which can be structured as one or more independent SIMD and/or MIMD machines. Here, an array of N PE's are controlled by Q microcontrollers (MC's). Each MC controls N/Q PE's. Thus, a maximum SIMD configuration of size N can be obtained by loading all the MC's with the same instruction simultaneously. Each PE has its own local memory which can be used for data storage in SIMD mode and for both data and instruction storage in MIMD mode. MC68000 processors are used as basic processor units. The full PASM system is envisaged to have N=1024 and Q=32.

* Pyramid Architecture

Hierarchically organized and locally interconnected pyramid structures (also called processing cones or layered recognition cones) support simple yet highly efficient image processing algorithms [Burt84]. Pyramids, in general, provide successively condensed information (e.g. reduced-resolution, fine-coarse approximation of some descriptive information) of the 'raw' images presented at the base layers. Thus, meaningless low-level iconic representations of the image are organized and converged into higher-level, meaningful, symbolic labels.

Pyramids have the advantages of parallel processing, pipelining, message passing (this reduces interprocessor distances from O(N) to O(log N) in

an NXN array). The pyramid being built by Boeing will have 7 array layers : 64X64, 32X32, 16X16, 8X8, . . .1X1, which will act in SIMD mode under the control of a central controller [Uhr85]. Design considerations for a pyramid machine in VLSI are discussed in [Dy82].

• **VLSI array processors**

Remarkable advances in VLSI circuitry have given the impetus to this new area of parallel architectures. But VLSI has its own constraints. *Components and interconnections are made of same 'stuff' and hence area is a uniform cost measure for both. Time of communication also depend upon the distance between the two* [Leis83]. Thus VLSI architectures stress on requirements like communication with nearest neighbours, less interconnection, less supervisory control, less I/O bandwidth between PE's and less overheads in communication.

Most of the signal processing algorithms have some common characteristics such as regularity, recursiveness, localized communication etc. Consequently VLSI architectures are becoming popular for signal and image processing applications. Systolic arrays and wavefront arrays are typical of such structures.

In a systolic array, all the PE's operate in synchronism with a global clock while *data flows from the main memory in a rhythmic fashion, passing through different PE's before it returns to the memory* [HTKung82]. Systolic array architectures are specially useful for compute bound problems, where the number of computations is greater than the number of I/O operations performed. This is achieved by ensuring that, once a data is accessed from the memory, it is used efficiently at each cell it passes while being propagated from cell to cell in the array. Wavefront arrays combine the concept of dataflow with

systolic computation thus alleviating the requirement of global synchronization in a systolic system.

Although systolic structures are well suited for VLSI implementation, high performance systems based on systolic principle have also been made using discrete components. 'Warp' processor developd at CMU [HTKung84] is a linear systolic array implemented with off-the-shelf IC's. Saxpy Matrix-1 is another example.

§ 1.3 *SYSTOLIC ARRAYS*

In this section we will describe the basic principles of a systolic array. A systolic array is a parallel special-purpose architecture made out of a number of processing elements of a few types called cells [Quin86]. Each cell performs some simple operation and the cells are interconnected in some regular pattern (e.g. linear, mesh, or hexagonal network). The basic principle of systolic computation rests on the fact that by replacing a single processing element with an array of cells, a higher computation throughput can be achieved without increasing the I/O bandwidth requirement. Following are the basic features of a systolic array.

- **Spatial regularity and locality**

The design must be made with finite type of processing elements which are locally interconnected. Use of buses for purposes other than clocking is not allowed, although it may be useful in certain situations.

- **Temporal regularity and synchrony**

Each cell should behave as a finite field automata, in synchronism with a global clock. But this does not prevent the cells from executing different programs at

different time instants.

• **Pipelinability**

If a systolic array has N cells, it must achieve a linear speed-up of O (N).

• **I/O proximity**

All input/output operations should be done with boundary cells only and no internal cell can access the outside world directly. Thus in a mesh connected array of N cells, the number of I/O operations must be O ($\sqrt{N}$).

• **Modularity**

Systolic arrays are designed for solving a particular class of problem, and the array can be extended simply by adding new cells to handle larger size problems. Application area can also be widened by adding cells of different types to take care of the additional functions that may be required by new problems.

The properties of a systolic array as described above account for a number of advantages of such systems. The property of spatial regularity divides the time required for developing a system by a regularity factor. Locality ensures avoidance of long capacitive wires. Temporal regularity and synchrony reduces the number of control lines. Pipelinability provides high performance. I/O proximity keeps the I/O bandwidth low. Finally, modularity allows design of a system according to the size of the problem.

§ 1.4 *OBJECTIVES AND SCOPE OF THE CURRENT WORK*

In this thesis, an attempt has been made to develop a systolic system for executing a class of digital signal processing algorithms. The objectives of

the current work are described below.

i) To define a systolic architecture for DSP applications which will work as an attached processor to an external host.

ii) To map some common algorithms onto the suggested architecture.

iii) To design and implement an interface cell for the array to support I/O requirements of the array as well as to provide independence to the systolic array from the host during execution of algorithms.

iv) To implement a simplified cell to test the working of the system.

- Chapter 2 describes the architecture of the Systolic Array Signal Processor (SASP) which has been partially implemented in this work. We will follow the array configuration that was proposed by Nemawarker [Nem88] and concentrate on the the architectural details of the functional units.
- Chapter 3 discusses the general mapping procedures to project conventional sequential algorithms onto a specified array structure.
- Mapping of LU and QR decomposition algorithms on SASP is given in Chapter-4.
- Chapter-5 details the design and implementation aspects of a part of the interface cell (remaining aspects are considered in the thesis [Us89]).
- Chapter-6 deals with the design of a simplified cell and discusses execution of a few algorithms on the cell in conjunction with the interface cell.
- We conclude with Chapter-7, noting down the experience gathered during the current work and suggesting some enhancements over the current design.

Chapter 2

THE 'SASP' ARCHITECTURE

§ 2.1 *INTRODUCTION*

Systolic array structures support high speed computing by effective use of a large number of processors. Although systolic systems follow some standards regarding interconnection patterns (e.g., localized communication) and a few structural features (e.g., pipelining of data and intermediate results), details of the architecture depends mainly upon the application requirements. This is because such systems are meant to be used as application specific processors attached to a general purpose computer.

Thus, the application needs give an important cue to the identification of a suitable architecture, and thus architectural optimization for a narrow set of algorithms can be done. Ease of implementation and programming can guide the choice. But at the same time, attempts should be made to balance some other apparently conflicting factors such as generality, flexibility and efficiency.

In this chapter, an architecture that attempts to satisfy the above requirements is proposed, and its main features are discussed.

§ 2.2 *SYSTEM ARCHITECTURE*

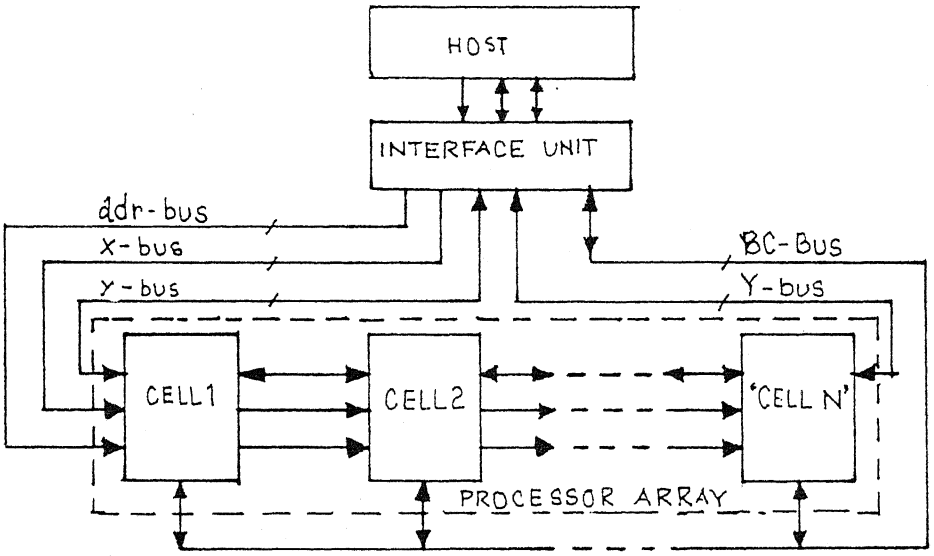


FIG. 2.1 SASP machine overview

The SASP machine (Fig.2.1) consists of two parts – processor-array (PA), and the interface unit (IFU).

i) *Processor-array* Main computing power of SASP stems from processor-array. It is a linear array of identical processors called **cells**. Each cell is a programmable microengine with a microcoded control unit of its own. The microcode for the cell can be downloaded by the host through the broadcast bus (**BC-bus**).

Each cell is connected to three input channels– **X-bus**, **Y-bus** and **adr-bus** ; apart from the **BC-bus**. Operand data are presented on **X-bus** and **Y-bus**. Generally **X-bus** carries the input data which is passed unchanged from one cell to the other in a pipelined fashion while **Y-bus** contains the partial output. Data on the **Y-bus** get updated as it pass through each cell, and final output is

obtained from the output of the last cell.

A local memory in each cell enables storage of data which are local to the cell and do not need to be transmitted (a typical example is matrix multiplication where each cell contains one column of the matrix, while, row elements are passed though the cells. Once the pipeline is full, each cell gets a row element in each cycle and multiplies it with corresponding column element stored in its local RAM). The address for the local memory also follows a regular pattern depending upon transmitted data (e.g., first element of rowi multiplies with first element of all the columns). Hence address can also be pipelined as data and this is done on **adr-bus**.

ii) ***Interface unit*** The interface handles all I/O operations between the host and the processor array. It receives all the input data from the host at a time, and supplies it to the processor array at proper time instants, and stores the output of computation. In this capacity the interface satisfies the I/O bandwidth requirement of the high speed array. In addition to this, the interface produces the address for the **adr-bus** and generates control signals on the **BC-bus** for loading data and microprogram into individual cells. For ease of description we will refer IFU as cell0, and cells of the **processor-array** as cell1..to...cellN. The IFU interacts with the host and array in two different states – host_state and array_state. These two modes will be discussed in detail in the next section.

§ 2.3 *STATES OF OPERATION OF THE 'SASP'*

As mentioned in the previous section, the SASP system operates in two

different states.

i) ***Host_state*** In this state, the interface allows the host to access the data memories and microprogram memories of the **IFU** (cell 0) and the processor cells. Host writes data and microcode onto respective cells before handing control over to the **IFU** and **processor-array**.

ii) ***Array_state*** After data and program has been written during host state; execution occurs in array_state . In this state, the host in denied access to the SASP machine, and the IFU controls the data flow. Note that, data on the Y-**bus** can flow in either direction. Accordingly, the IFU treats data flow in two different configurations.

a) *Forward mode* In this mode, Y-data is fed to the leftmost cell (cell 1) and output is taken from the output bus of the rightmost cell (cell N).

b) *Reverse mode* In this mode, Y-data is fed to cell N and output is taken out from cell 1.

In both the confoigurations, X-data is fed to cell 1. Y-data can be circulated into the array through the interface.

§ 2.4 *INTERCELL COMMUNICATION*

In SASP architecture, global communication requirements are greatly reduced. **BC-bus** is required only during loading of microprogram and data. But once execution starts, only local communication channels between cells are used. Because of this regular and highly local communication pattern, the processor-array can be easily augmented using additional cells.

Each cell communicates only with its left and right neighbour.

Communication is done through three channels as described above. A one way protocol is used for data transfer. Each cell has an input queue on each channel to buffer the data stream. When $cell_n$ sends data to $cell_{n+1}$ along any of the three channels, it also sends the write signal to the corresponding input queue of $cell_{n+1}$. So it is the responsibility of $cell_n$ to write the data into the input queue of $cell_{n+1}$ before operation at $cell_{n+1}$ can take place. If the data is not written into the queue before $cell_{n+1}$ tries to read it, a queue empty signal is generated which blocks all operations of $cell_{n+1}$ until the data arrives. This blocking is transparent to the operation of the array in the sense, that other unblocked cells continue to operate as usual. Similarly blocking can occur while writing onto a queue which is already full. The cell sending data would remain blocked until a data is read from the full queue by the respective cell, thus, providing a free location to write data.

§ 2.5 *SIGNALS ON BROADCAST BUS*

The broadcast bus (BC-bus) is used by the host to load data and microprogram into individual cells. This bus originates at the interface and connects all the cells of the array. Following are the signals that are present on the BC-bus.

- BCD_0—BCD_7 These are 8-bit data lines which contain data or microcode to be loaded to a particular cell. The SASP architecture is proposed for 32- bit floating point operation and hence bus width of 32 bits would be most desirable. But in the current work, we were constrained by limited board area and therefore, we used 8-bit data buses in oreder to limit the chip count and

thereby saving limited board space.

- BCA_0-BCA_3 They contain a 4-bit address to specify a particular cell. Most of the lines of the BC-bus enter the cells through a transparent latch which is strobed only when the address on these lines matches the address of the cell. At present, a maximum of fourteen cells ($=2^4-2$) can be connected to the array. One address is reserved for the IFU and one is used as broadcast address. A broadcast address is provided to speed up the loading process in case all the cells do the same function and therefore, use the same microprogram.
- Lμc This line is made high when a microcode read/write is intended. During execution this signal must be made zero.
- Lwt This line is made high for reading/writing data from/to the cells. This should also be made zero during execution.
- $\overline{Clr}$ Clears the counter that selects different banks of microcode RAMs (μR0, μR1 etc. See § 5.6 for details).
- $\overline{Wcs}$ It is asserted each time a microcode loading process is started at a particular microcode memory bank. Putting a zero on this line puts the microengine of the addressed cell in 'writable control' mode which will be discussed in detail in chapter5.
- RST This is a software reset signal used to reset the microengine of a particular cell (Chapter5).
- SYNC It is used to trigger all the cells into execution at the same system clock cycle in a synchronous manner.
- $\overline{Wr\mu c}$ This is the write pulse to write data or microcode in their respective RAMs.
- $\overline{Rd\mu c}$ This takes care of reading of data or microcode.

- Flg This is a handshake signal for microcode read/write and will be explained in § 5.4.
- ClkB This global clock is given to all the cells and the whole array operates in synchronism with this single clock.

§ 2.6 *FEATURES OF 'SASP' ARCHITECTURE*

The linear array structure of SASP array simplifies interconnections and data flow control to a great extent. Because of locality and regularity of interconnections, the array is expandable by addition of new cells and no extra modification has to be done. For an extremely large size array, global synchronization can cause a problem, but with the facility of run time flow control described in § 2.4, a switchover to wavefront principle can be easily done. Application range of SASP can be widened quite easily. An algorithm, currently not supported by SASP because of limited computation power of the cells, can be brought under its purview by adding an upgraded cell to the existing array and by mapping the algorithm in such a manner, that the special computation demanded by the new algorithm is confined to the new cell only. Since each cell of the SASP array is programmable, it can also be used as a SIMD or MIMD machine to execute non systolic algorithms.

Chapter 3

MAPPING ALGORITHMS ONTO SYSTOLIC ARRAYS

§ 3.1 *INTRODUCTION*

Algorithms are typically written in sequential code. This causes an ordering in the program which may not be required at all, as far as the final output is concerned. For example, let us consider the mathematical expression to compute the addition of two matrices A and B.

C = A +B ; $c_{ij} = a_{ij} + b_{ij}$

The corresponding FORTRAN code can be written as,

```
      DO  10  j = 1 to N
      DO 10   i = 1 to N
      C (i,j) = A (i,j) + B (i,j)
10    CONTINUE
```

Although all the elements C (i,j) can be computed simultaneously, the above code instructs the sequence of operations in a column major order. This ordering feature of sequential codes doesn't consider the parallelism inherent in certain operations performed, and therefore, sequential codes are not fit for algorithms intended to be executed on a parallel machine.

Systolic array is a member of a special class of parallel machine known as array processors. Its distinctive characteristics are – local communication, balanced distribution of load among processing elements, pipelining, regularity, modularity etc. To process an algorithm on a systolic array, these features of the machine must be exploited so that, the algorithm can be executed efficiently on it. A number of different methods have been proposed in this direction to explore the data dependency and parallelism of an algorithm [Moldo83, Sykung87, Leis83], and to match it to the required array topology. In this chapter, a simple yet quite generalised approach as proposed in [Sykung88] will be described through mapping of some common signal processing algorithms.

§ 3.2 *DESCRIPTION OF ALGORITHMS*

This is an important step in an algorithm oriented design. The ultimate design should begin with a powerful algorithmic notation that is easy to understand and at the same time expresses the recurrence and parallelism associated with the algorithm.

There are many different expressions that can be used to represent a parallel algorithm such as snapshots, recursive algorithms with space time indices, parallel codes, dependence graphs (DG's) etc. A VLSI algorithm is often very regular and the computation activities are conveniently expressed by a simple grid model called DG and from the DG representation, the array configuration can be easily determined. We will follow the Dependence Graph approach here.

The first step towards designing the Dependence Graph is to convert the given sequential description of the algorithm into a form called single assignment code:

" A single assignment code is a form where every variable is assigned only one value during the execution of the algorithm. " [Sykung88].

Example 1. LINEAR CONVOLUTION ALGORITHM

Convolution of two sequences x_j and w_j is defined as

$$y_j = \sum_{k=0}^{j} x_k * w_{j-k} = \sum_{k=0}^{j} x_{j-k} * w_k$$

If u is of length N, and w is of length P then j = 0, 1, 2, 3, 4,. . . . . , N+P-2

Note, that in the above expression y_j is overwritten many times and thus assigned more than once. The single assignment version of the algorithm becomes

$$y_j^k = y_j^{k-1} + x_j^k * w_j^k \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 3.1$$

where $y_j^0 = 0$; $x_j^k = x_{j-1}^k$; $x_o^k = x_k$

$w_j^k = w_{j-1}^{k-1}$; $w_j^0 = w_j$

Here k can be viewed as time index and j as space index. But from the mathematical point of view, they can be treated equally, and, j and k together make the index space of the algorithm.

Example 2. AR FILTERING ALGORITHM

It is described by the following relationship,

$$y_j = \sum_{k=1}^{N} a_k * y_{j-k} + u_j$$

The single assignment form can be given in two ways :

i) $y_j^k = y_j^{k-1} + a_j^k * y^N_{j-k} \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 3.2$

where $y_j^0 = u_j$; $a_j^k = a_{j-1}^k$; $a_{-1}^k = a_k$ and $y_j^N = y_j$

ii) $y_j^k = y_j^{k+1} + a_j^k * y_{j-k}^1 \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 3.3$

where $y_j^{N+1} = u_j$; $a_j^k = a_{j-1}^k$; $a_{-1}^k = a_k$ and $y_j^1 = y_j$

§ 3.3 *DEPENDENCE GRAPH (DG)*

A dependence graph (G = (N,A)) represents graphically the computations that occur in an algorithm, where N is a set of *nodes* which represent the computations and A is a set of *arcs* representing dependency between computations. An arc whose endpoints are the same is called a *loop*. A *chain* is a sequence of arcs ($a_1 \ldots a_n$) such that arc a_r ($2 \leq r \leq n-1$) has one endpoint common with arc a_{r-1} and its second endpoint common with arc a_{r+1} regardless of the direction of the arcs. A *path* is a chain formed with arcs which are directed the same way. A *cycle* is a path whose endpoints are the same.

It is to be noted that, a node represents an operation but the details are deliberately ignored. A DG can be computable if it does not contain any loop or cycle. A *shift invariant* DG is one in which the dependency arcs connected to all nodes are independent of their positions. In the following discussions, all the DG's are assumed to be shift invariant unless otherwise specified. However, in array design, the I/O nodes (boundary nodes) are exempted from this requirement.

§ 3.4 *MAPPING METHODOLOGY*

§ 3.4.1 *DESIGN STEP 1. MAPPING ALGORITHM ONTO DG*

An algorithm is first written in single assignment form. The space-time index space of the algorithm corresponds to the lattice space of the DG. The data dependencies are shown very clearly in single assignment form, and those dependance relations can easily be translated onto DG by connecting

corresponding nodes. The DG for convolution algorithm and AR filtering algorithm are shown in Fig 3.1a & 3.1.b respectively.

§ 3.4.2 *DESIGN STEP 2 : MAPPING DG ONTO SFG*

Once the DG has been formed, it is possible to design the array by assigning a PE to each of the nodes. But this will prove highly inefficient since each PE will be operating only for a small fraction of the total computation time. To utilize the PE's more efficiently, the DG, therefore, should be mapped in a more compact intermediate form called SFG.

Signal flow graph (SFG)

This is also a graphical representation of the algorithm concerned, but it gives a more concise and specific description of the operations, as compared to DG. Unlike DG, SFG has a functional part which specifies operations performed at each node. As in DG, the structural part of SFG consists of vertices V representing computations, and a set of edges E indicating data dependencies; in addition, there one more set D(E) which gives number of delays associated with each edge.

The operations at the nodes are assumed to be performed in zero time and any delay in computation is shown explicitly on dependency arcs. This eases the analysis of space time activities associated with pipelining. In contrast to DG, an SFG can contain loops or cycles as long as there is at least one delay in that loop or cycle.

SFG projection procedure

This consists of two parts :

i) *Processor assignment* : This step is concerned with the problem of

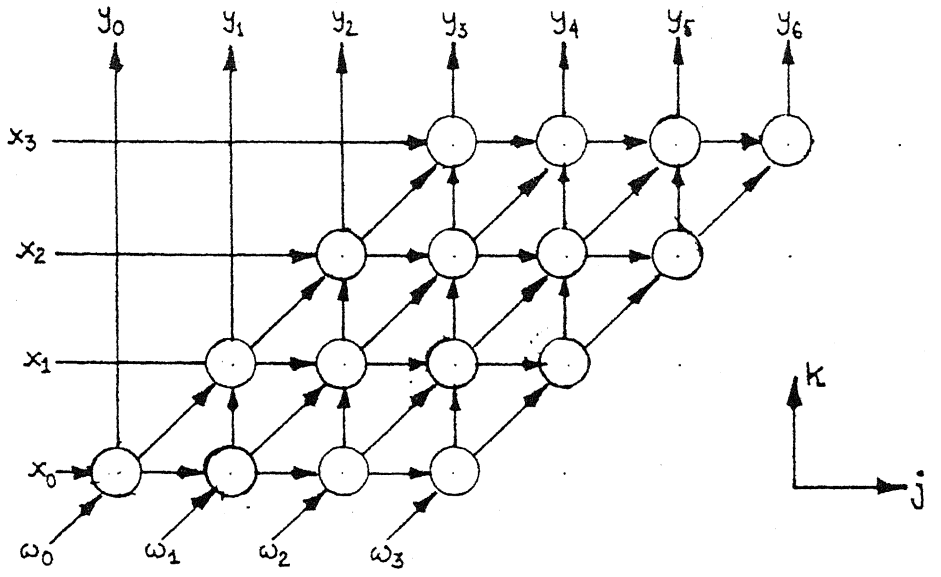


Fig. 3.1.a DG FOR CONVOLUTION

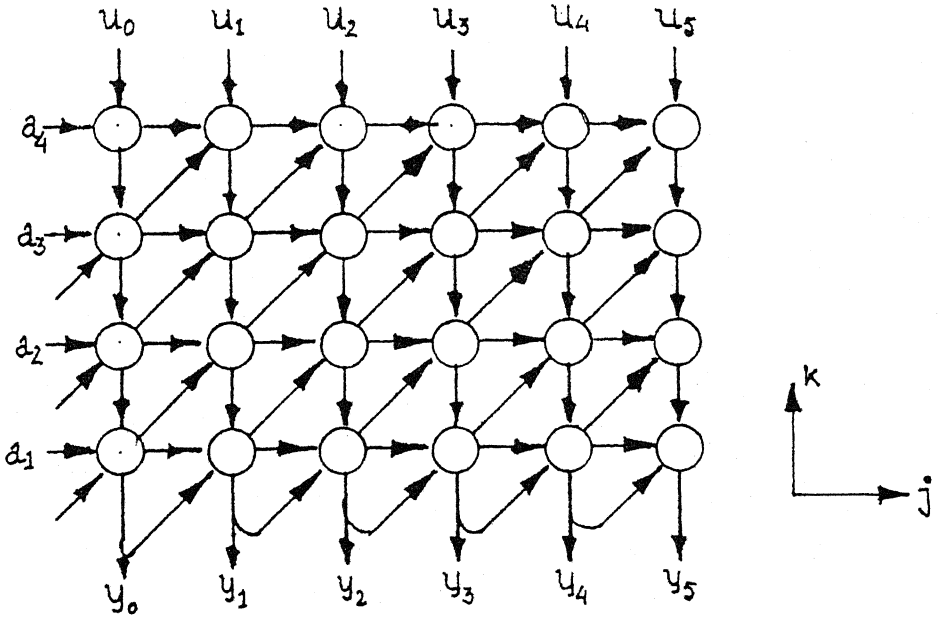


Fig. 3.1. b DG For AR-Filtering

assigning an operation to a particular processor. Since algorithms generally implemented on systolic arrays are characterized by high degree of regularity, a linear projection for processor assignment is almost sufficient. But for algorithms which involves irregular data-dependencies, non-linear assignment may be necessary. Standard signal processing algorithms are quite regular and hence, only linear projection assignment will be treated here.

In this type of processor assignment, all the nodes of the DG in a straight line are projected to a single node in the resulting SFG. The projection direction is denoted by the vector $\vec{d}$.

ii) *Scheduling* : The scheduling scheme specifies the order in which operations are sequenced in all the nodes of the SFG. The scheduled execution time of a node is denoted by a time index. A linear schedule, denoted by $\vec{s}$ maps a set of pareallel, uniformly placed, equitemporal hyperplanes of the DG to e set of linearly increased time indices ($\vec{s}$ is the vector normal to the equitemporal hyperplanes). The time index of a node $\vec{i}$ can be represented by $\vec{s}^T . \vec{i}$ where $\vec{i}$ is the index of the node.

To obtain a valid SFG from a given DG the projection direction $\vec{d}$ and the schedule vector $\vec{s}$ have to satisfy two conditions as given below.

i) Condition for causality : The permissible schedule vector must obey the data dependency of the DG, i.e. if node i depends upon the output of node j , then j must be scheduled ahead of i. This can be described by the following expression ;

$$\vec{s}^T . \vec{e} \geq 0 \quad \text{for any dependency arc } \vec{e} \qquad \ldots\ldots\ldots\ldots\ldots\ldots 3.4$$

ii) Condition for parallelism : If $\vec{s}$ and $\vec{d}$ are orthogonal to each other i.e. if $\vec{d}$ is parallel to the equitemporal hyperplanes, then sequential processing will

result. Thus the condition for parallelism can be described mathematically as :

$$\vec{s}^T \cdot \vec{d} > 0 \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots 3.5$$

The mapping process from DG to SFG can be summarized mathematically as follows.

i) *Node mapping* : The mapping of a node $\vec{i}$ of the DG onto a node $\vec{j}$ in the SFG is found by

$$\vec{j} = P^T \cdot \vec{i}$$

where P is a NX (N−1) matrix orthogonal to $\vec{d}$ i.e. $P^T \cdot \vec{d} = 0$

It is to be noted that the mapping method maps an N dimensional DG onto an (N−1) dimensional SFG.

ii) *Arc mapping* : This maps the arcs of the DG to the edges of the SFG and also gives the delay D ($\vec{e}$) associated with each edge. For a given arc $\vec{a}$, corresponding D ($\vec{e}$) and $\vec{e}$ can be determined by

$$\begin{bmatrix} D(\vec{e}) \\ \vec{e} \end{bmatrix} = \begin{bmatrix} \vec{s}^T \\ P^T \end{bmatrix} * \begin{bmatrix} \vec{a} \end{bmatrix}$$

iii) *I/O mapping* : Since, I/O node of the DG do not satisfy the condition of shift invariance, special treatment is required for these nodes. The SFG node position $\vec{j}$ and I/O time t ($\vec{i}$) of an I/O of the DG node $\vec{i}$ can be derived as;

$$\begin{bmatrix} t(\vec{i}) \\ \vec{j} \end{bmatrix} = \begin{bmatrix} \vec{s}^T \\ P^T \end{bmatrix} * \begin{bmatrix} \vec{i} \end{bmatrix}$$

Example 1 SFG FOR CONVOLUTION

The DG shown in Fig 3.1.a can be projected in several different ways by choosing different directions of projection. If we choose to project along j axis

i.e. with $\vec{d} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, a permissible linear schedule $\vec{s}$ would be $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$, although there can be other valid choices. The processor basis P^T would be $\begin{bmatrix} 0 & 1 \end{bmatrix}$.

i) *Node mapping* :

$$\vec{j} = \begin{bmatrix} 0 & 1 \end{bmatrix} * \begin{bmatrix} j \\ k \end{bmatrix} = \begin{bmatrix} k \end{bmatrix}$$

ii) *Arc mapping* :

$$\begin{bmatrix} D(\vec{e}) \\ \vec{e} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

iii) *Input mapping* :

$$\begin{bmatrix} t(\vec{i}) \\ \vec{j} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} j & 0 \\ 0 & k \end{bmatrix} = \begin{bmatrix} j & k \\ 0 & k \end{bmatrix}$$

iv) *Output mapping* :

$$\begin{bmatrix} t(\vec{i}) \\ \vec{j} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} j \\ 3 \end{bmatrix} = \begin{bmatrix} j+3 \\ 3 \end{bmatrix}$$

The resulting SFG is shown in Fig 3.2.a.

Example.2 SFG FOR AR FILTERING ALGORITHM

$$\vec{d}^T = \begin{bmatrix} 1 & 0 \end{bmatrix} \quad ; \quad \vec{s}^T = \begin{bmatrix} 1 & 0 \end{bmatrix} \quad ; \quad P^T = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

i) *Node mapping* :

$$\begin{bmatrix} 0 & 1 \end{bmatrix} * \begin{bmatrix} j \\ k \end{bmatrix} = \begin{bmatrix} k \end{bmatrix}$$

ii) *Arc mapping* :

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix}$$

iii) *I/P mapping* :

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} j \\ 3 \end{bmatrix} = \begin{bmatrix} j \\ 3 \end{bmatrix}$$

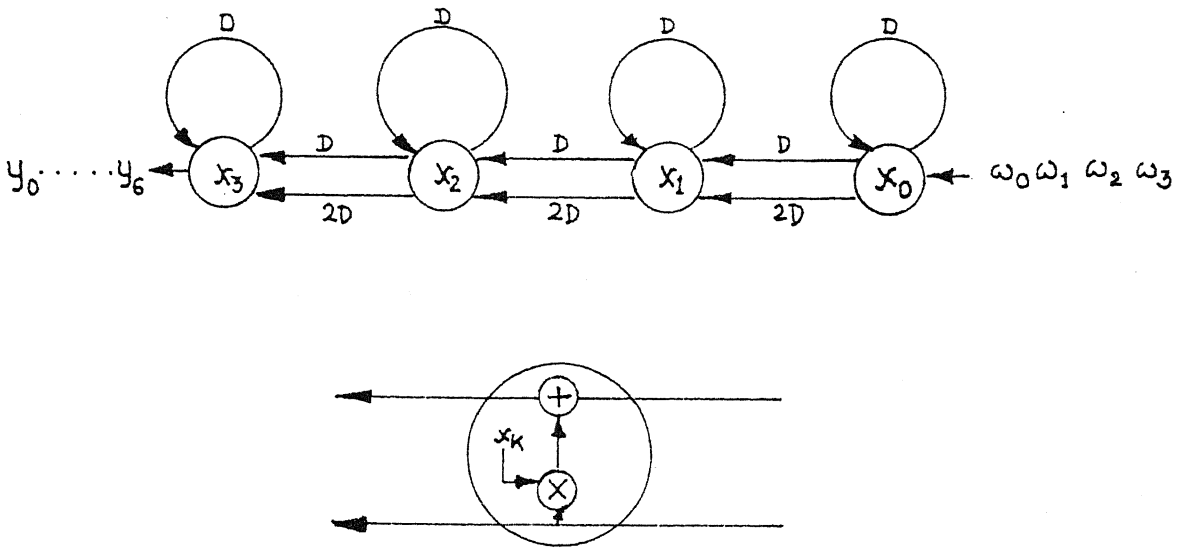


Fig. 3.2.a SFG For Convolution

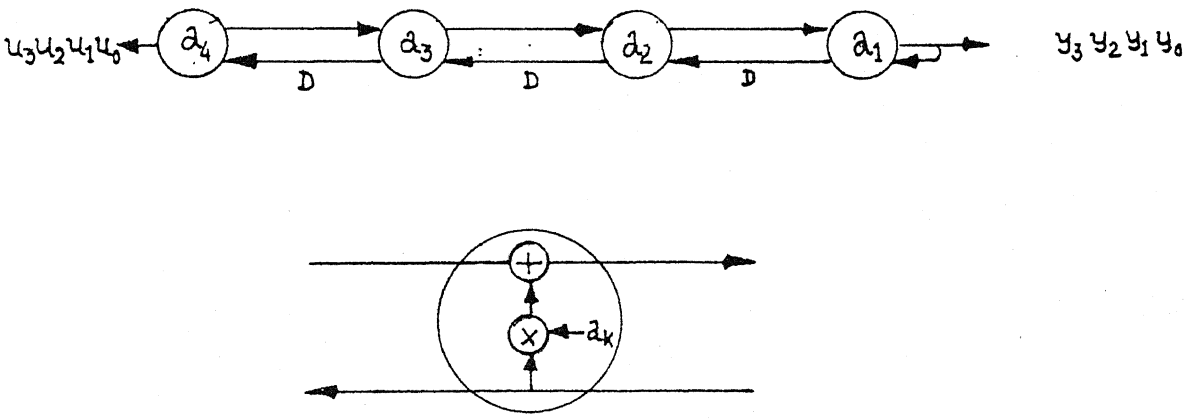


Fig. 3.2.b SFG For AR-Filtering

iv) *O/P mapping* :

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} j \\ 3 \end{bmatrix} = \begin{bmatrix} j \\ 3 \end{bmatrix}$$

The SFG is shown in Fig 3.2.b.

§ 3.4.3 *DESIGN STEP 3. MAPPING SFG ONTO SYSTOLIC ARRAY*

The SFG obtained in step2 can be realised by a multiprocessor array. For an algorithm having a DG of N dimensions, step2 of the mapping procedure gives an SFG of dimension N-1. Direct array implementation will require an array of the same dimension which again may not prove cost effective. Solution to this end can be found by going through step2 repeatedly; thereby, reducing the SFG dimension by one in each iteration, until the resulting SFG achieves a feasible dimension. This process can be carried out only as long as the SFG remains shift invariant. Furthermore, extra contraints different from those mentioned in step 2 would also have to be satisfied. This will be demonstrated in Chapter-4, with reference to the mapping of LU-decomposition algorithm onto a linear systolic array.

When the SFG achieves the desired dimension, a little retiming is required to convert it to a systolic array. The SFG itself represents an array structure which differs from the systolic array architecture in the following aspects.

i) Broadcasting of data is allowed in SFG, but not permitted in systolic array.

ii) Even if the data is spatially localized in an SFG, it may not be temporally localized, as in a systolic array. Mathematically, a permissible systolic schedule is slightly more demanding than a valid SFG schedule (eq. 3.4 & 3.5). They are

listed below:

$$\vec{s}^T \cdot \vec{e} > 0 \qquad \text{and} \qquad \vec{s}^T \cdot \vec{d} > 0$$

The need for temporal locality can be alleviated easily by localizing the SFG. For example, on the index space (i,j) of the SFG, let a data a_{i_1} is broadcast over all nodes having index (i_1, j). This can be avoided by writing the variable a_{i_1} in the following manner:

$$a_{i_1}{}^0 = a_{i_1} \qquad \text{and} \qquad a_{i_1}{}^{j+1} = a_{i_1}{}^j$$

If j is a spatial index then, the data a_{i_1} is localized spatially but not temporally, which is not permissible in systolic array. Temporal locality can be achieved by introducing additional delay and it has been proved [SYKung88] that any SFG can be temporally localized by this method. This method of systolizing an SFG is explained below with the example of AR filtering.

The SFG for AR filtering (Fig 3.2.b) does not satisfy the condition of temporal locality since there are edges with zero delay. It can be temporally localized by 'cut-set retiming ' which is based on two rules.

Rule 1. *Time scaling* : All delays can be scaled i.e. $D \rightarrow \alpha D'$ by a single positive integer which is also known as the pipelining period. Correspondingly, inputs and outputs also have to scaled.

Rule 2. *Delay Transfer* : For any cut set of the SFG (a cut set divides the SFG in two different parts), the edges of the cut set can be grouped into inbound and outbound edges depending upon the directions of the edges. According to *Rule* 2, each outbound edge can be advanced by KD' time units while each inbound edge can be delayed by the same amount. For a time invariant SFG, the general system behaviour is not affected because effects of advances and lags cancel each other:

Systolization of Fig 3.2.b is done in two steps.

i) Time scaling : Let us scale the delay D by a factor 2 i.e. $D \rightarrow 2D'$ (Fig 3.3.a).

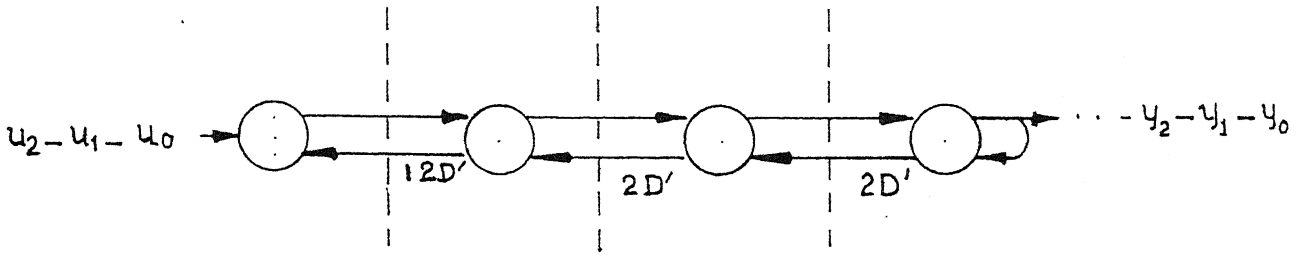
ii) Delay Transfer : Applying cuts uniformly along the dotted lines in Fig 3.3.a, a delay D' can be added to each rightgoing edge, and to compensate this, a delay D' can be subtracted from every leftgoing edge. The resulting SFG is shown in Fig 3.3.b.

Now, that the SFG has been temporally localized, it is ready to be converted into systolic array. For this, the only requirement is to introduce a delay into each of the operating modules. This delay, combined with the zero delay operation of a node forms a basic processor element or PE. Remaining edge-delays are pure delay elements. Since self loops are implemented as registers in the PE, they are also combined into the PE. Fig 3.4 gives a detailed illustration of this.

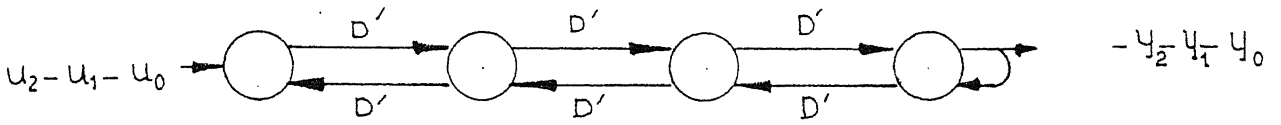
The systolic arrays for convolution algorithm and AR filtering algorithm can now be easily found as shown in Fig 3.5.a and 3.5.b respectively.

§ 3.5 *LARGE SIZE PROBLEMS*

So far, the discussion of mapping technique has been done with an implied assumption that an array has sufficient number of PE's to execute the algorithm in a single pass. The advantages of systolic or any kind of parallel computing are perceptible only when the problem size is sufficiently large. In practice, it is not possible to increase the size of the array beyond a certain



3.3.a.



3.3.b.

Fig. 3.3 Systolization of SFG for AR-Filtering by , a) Time Scaling

b) Delay Transfer.

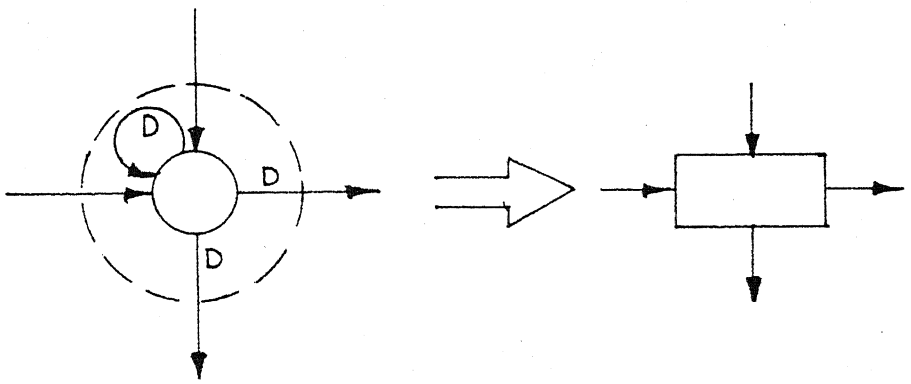


Fig. 3.4 Illustration of Construction of a PE from a Node in SFG.

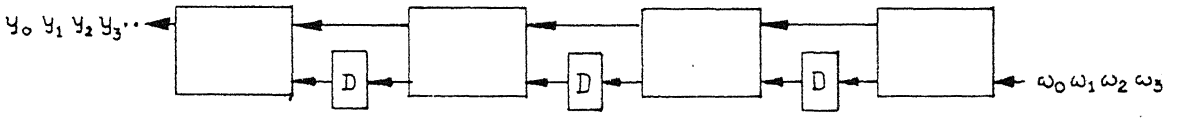


Fig. 3.5 a) Systolic Array for Linear Convolution

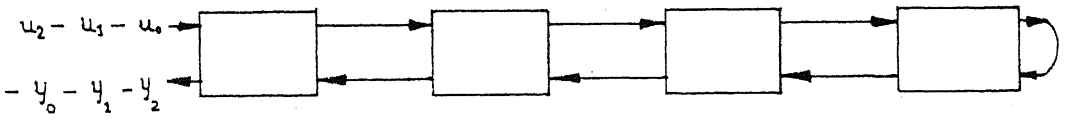


Fig. 3.5 b) Systolic Array for AR-Filtering.

limit and , therefore, if the problem size is large compared to the physical size of the array, the problem should be partitioned into a number of subproblems which can be directly mapped onto the given array size.

Consider a DG having N dimensions and let it be projected onto an SFG of dimension N-1 and of size $s_1 X s_2 X s_3 \ldots . X s_{N-1}$. The SFG is partitioned into $n_1 X n_2 X n_3 \ldots . X n_{N-1}$ blocks, where each block is of size $d_1 X d_2 X d_3 \ldots . X d_{N-1}$.

Clearly, for i = 1 . . . . N-1, $d_i = s_i / n_i$.

There are two methods for mapping the SFG onto an array :

i) *Coalescent* or *locally sequential globally parallel* (LSGP) method.

In this method, each block is mapped into a PE. Each PE sequentially executes the nodes of the corresponding block, while all the blocks get executed parallely in different PE's. Hence, the number of blocks should be equal to the number of PE's in the array. This scheme requires that each PE has sufficient local memory to store the node data.

For a given N dimensional DG, and a projection direction $\vec{d}$, let the processor basis be P. For an acceptable schedule $\vec{s}$, the $d_1 X d_2 X d_3 \ldots . X d_{N-1}$ nodes residing in a block should be executed at different times, so that, at any instant, at most one node is active in each block [SYKung88, Navar87].

ii) *Cut and pile* or *locally parallel globally sequential* (LPGS) method.

In this scheme, the blocksize is chosen to match the array size and all nodes in a block are executed concurrently (locally parallel). After one block has been finished, data for the next block is fed, and thus blocks are processed in a sequential manner. In this method, the memory requirement of each PE is

constant and independent of the size of the problem.

The LPGS method calls for careful node assignment. Since blocks are executed sequentially, they can not have any reverse data dependency between each other. Essentially, in this method, the algorithm is partioned by a transformation $\begin{bmatrix} \pi \\ \pi_p \end{bmatrix}$; where π_p divides the index set into a number of blocks, and π determines the sequence of operations inside the blocks [Moldo86].
A comparative pictorial illustration of LSGP and LPGS methods is given in Fig 3.6. Besides the above methods, some other techniques also have been proposed . They are briefly discussed below.

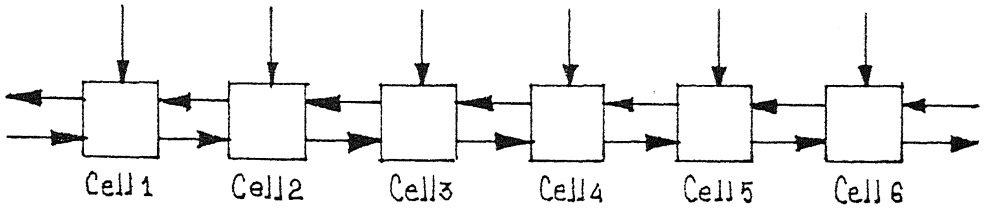
i) *Dense to band matrix transformation* (DBT) [Navar87].

This method is particulary suitable for matrix computations. Any matrix can be viewed as a band matrix of appropriate bandwidth. In band matrix approach, a matrix computation can be performed in a single pass through a fixed size array, as long as the bandwidth of the matrix is less than or equal to the array size. Hence, the size of the problem depends upon the bandwidth of the matrix and not upon its dimension.

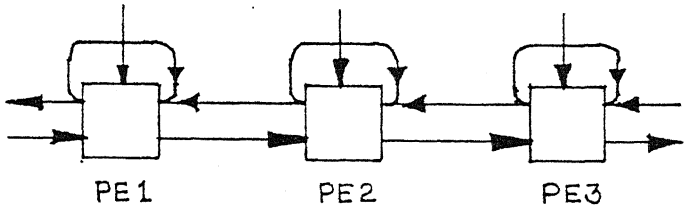
In DBT method a large matrix is first transformed into a band matrix of larger dimension, but having a bandwidth equal to the array size. Then the matrix can be computed on the fixed size array.

ii) *Algebraic partitioning*

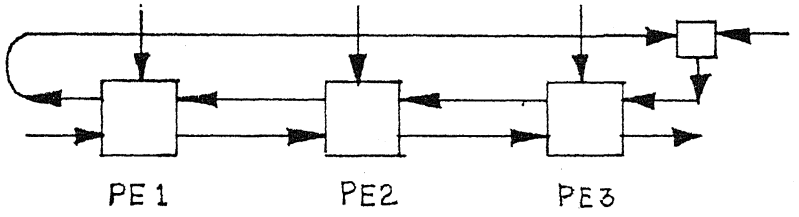
In this approach, a big problem is first decomposed algebraically into a number of smaller subproblems. For example, a 2D-convolution can be expressed by sums of products of different 1-D convolutions. The details of this scheme can be found in [SYKung88].



3.6.a.



3.6.b.



3.6.c.

Fig. 3.6 a) A Problem size Systolic Array that has to be Mapped onto a 3-Cell Linear Array.

b) LSGP Mapping

c) LPGS Mapping.

Chapter 4

MATRIX COMPUTATIONS ON SYSTOLIC ARRAY

§ 4.1 *INTRODUCTION*

Many scientific and technical applications require high computing speed; a large number of them involve matrix computation. Systolic array structures are quite amenable to matrix computations and this has triggered a host of activities to find array structures for different matrix algorithms. As a result, we have various linear array models to solve triangular linear systems, compute convolution of two sequences, perform AR filtering etc. All these algorithms can be executed directly on the linear stucture of SASP (Chapter-2). Two dimensional models to compute matrix multiplication, LU-decomposition, QR factorization etc. are also available with a number of choices. To implement these mesh algorithms on a linear systolic array, careful mapping is required. In this chapter, mapping of LU and QR decomposition on a linear systolic array will be discussed. Mapping of matrix multiplication is simple and can be done without going through systematic design procedures step by step and this will be shown later in conjunction with execution of algorithms on a single systolic cell.

§ 4.2 *MAPPING OF LU – DECOMPOSITION*

In LU-decomposition, a matrix A decomposes into an upper triangular and a lower triangular matrix. To get an unique factorization, this process can be broken into two steps.

1. $A = LDU'$

where,

L is a lower triangular matrix with . . . . $l_{ij} = 1$ if $i = j$ & $l_{ij} = 0$ if $i < j$

U' is an upper triangular matrix with . . $u'_{ij} = 1$ if $i = j$ & $u'_{ij} = 0$ if $i > j$

D is a diagonal matrix

2. $A = LU$

where,

$U = DU'$, which again is an upper triangular matrix. This ensures uniqueness of both U and L.

Now, $a_{ij} = \sum_{p=1}^{N} l_{ip}\, d_{pp}\, u'_{pj}$ where $i = 1. . . . . N$ & $j = 1. . . . . N$

but, as per definition, $l_{ij} = 0$ for $i < j$ and $u_{ij} = 0$ for $i > j$

Hence, $a_{ij} = \sum_{p=1}^{\min(i,j)} l_{ip}\, d_{pp}\, u'_{pj}$

for $i = j$, $\quad a_{ii} = d_{ii} + \sum_{p=1}^{i-1} l_{ip}\, d_{pp}\, u'_{pj}$

or, $\quad d_{ii} = a_{ii} - \sum_{p=1}^{i-1} l_{ip}\, d_{pp}\, u'_{pj}$

and, $\quad d_{11} = a_{11}$

for $i < j$ $\quad a_{ij} = d_{ii}\, u'_{ij} + \sum_{p=1}^{i-1} l_{ip}\, d_{pp}\, u'_{pj}$

or, $$u_{ij} = d_{ij}\, u'_{ij} = a_{ij} - \sum_{p=1}^{i-1} l_{ip}\, d_{pp}\, u'_{pj}$$

or, $$u_{ij} = a_{ij} - \sum_{p=1}^{i-1} l_{pj}\, u_{pj}$$

for $i > j$, $$a_{ij} = l_{ij}\, d_{jj} + \sum_{p=1}^{j-1} l_{ip}\, d_{pp}\, u'_{pj}$$

or, $$l_{ij} = \left[a_{ij} - \sum_{p=1}^{j-1} l_{ip}\, d_{pp}\, u'_{pj} \right] / d_{jj}$$

or, $$l_{ij} = \left[a_{ij} - \sum_{p=1}^{j-1} l_{pj}\, u_{pj} \right] / u_{jj}$$

The relations can be represented by the following recursive algorithm.

$$a_{ij}^{1} = a_{ij}$$

$$a_{ij}^{k+1} = a_{ij}^{k} - l_{ik}\, u_{kj}$$

$$l_{ik} = 0 \qquad \text{if } i < k$$

$$= 1 \qquad \text{if } i = k$$

$$= a_{ik}^{k} / u_{kk} \qquad \text{otherwise.}$$

$$u_{kj} = 0 \qquad \text{if } j < k$$

$$= a_{kj}^{k} \qquad \text{if } j \geq k$$

Before mapping this algorithm onto a systolic array, a single assignment description of the above will be helpful. This is obtained as follows. .

Single assignment form :

for, k from 1 to N

i from k to N

j from k to N

$$u_{kj}^{i} = a_{ij}^{k} \quad \text{if } i = k$$
$$= u_{kj}^{i-1} \quad \text{otherwise.}$$

$$l_{ik}^{j} = a_{ij}^{k} \,/\, u_{kj}^{i} \quad \text{if } j = k$$
$$= l_{ik}^{j-1} \quad \text{otherwise.}$$

$$a_{ij}^{k+1} = a_{ij}^{k} - l_{ik}^{j}\, u_{kj}^{i}$$

where, $a_{ij}^{1} = a_{ij}$; $u_{kj}^{N} = u_{kj}$ and $l_{ik}^{N} = l_{ik}$

The different steps of mapping are described below.

Step 1. DG design

The DG for this algorithm can be obtained by comparing the index differences between each assignment statement. The DG is shown in Fig 4.1.a.

Step 2. SFG design

The DG shown in Fig 4.1.a can be projected onto a two dimensional lattice in a number of ways. Choice of a particular direction depends upon the array structure on which the algorithm is to be executed. In case of SASP we are guided by the following requirements.

i) All I/O operations should be done through boundary nodes only.

ii) Any special function (e.g. division, square root) should be confined to a particular node rather than being distributed over all the nodes. This enables

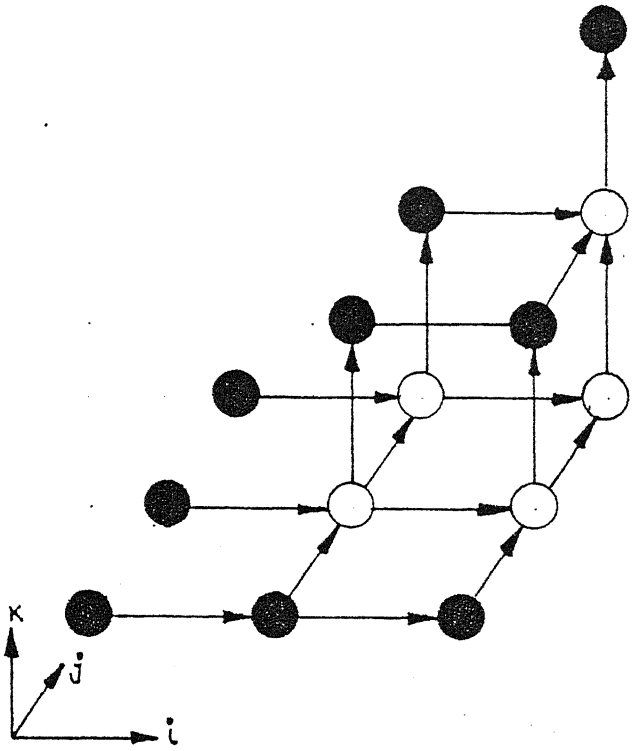


Fig. 4.1.a) DG for LU Decomposition

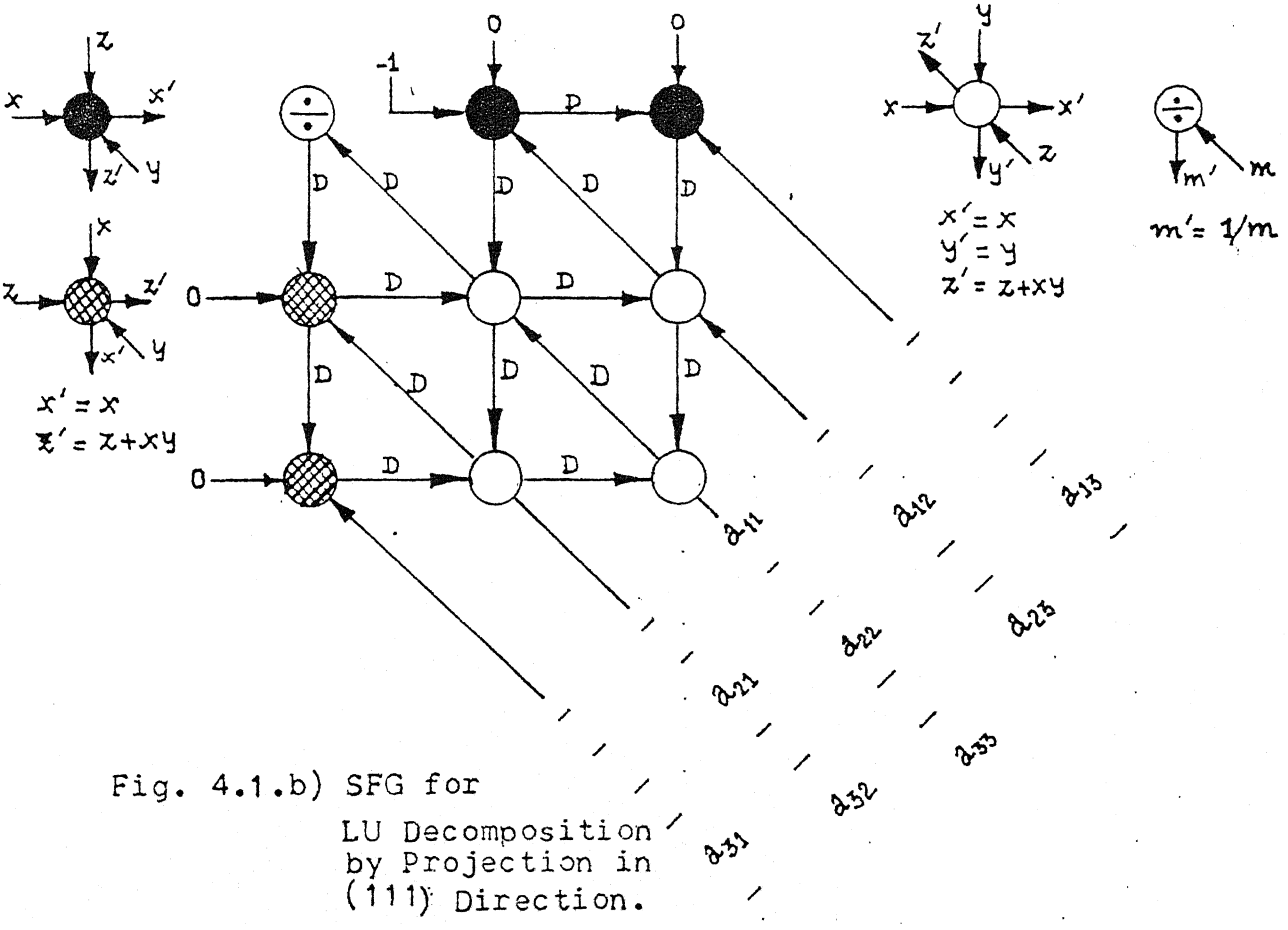


Fig. 4.1.b) SFG for LU Decomposition by Projection in (111) Direction.

handling of the special functions by addition of a new cell to the already existing array.

It is found that, for mapping onto SASP, projection along the vector $\vec{d}^T = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$ appears suitable. Next, we choose the transformation matrix $P^T = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix}$ such that $P^T \vec{d} = 0$.

i) *Node mapping* :

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} * \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i-k \\ j-k \end{bmatrix}$$

With a schedule vector $\vec{s}^T = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$, we find the other mappings as follows :

ii) *Arc mapping* :

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix}$$

iii) *Input mapping* :

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} * \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} = \begin{bmatrix} i+j+1 \\ i-1 \\ j-1 \end{bmatrix}$$

iv) *Output mapping* :

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} * \begin{bmatrix} i & k \\ k & j \\ k & k \end{bmatrix} = \begin{bmatrix} i+2k & j+2k \\ i-k & 0 \\ 0 & j-k \end{bmatrix}$$

The projected SFG thus obtained is shown in Fig 4.1.b.

Step 3. Projection of SFG

To get a linear array, the SFG obtained in step 2. has to be projected once more. Before doing this, some additional care has to be taken.

i) Projection in any direction of the SFG, shown in Fig 4.1.b, would result in input operation to interior cells. To avoid this, the index space of the SFG has to be extended. This is shown in Fig 4.2.a.

ii) Outputs of the SFG, shown in Fig 4.1.a are stored in respective nodes. They can be taken out at a time, after execution of the entire algorithm has been finished; or they can be outputted intermittently everytime an output element gets computed. The second method reduces the bandwidth requirement at the output, but it also reduces the scope of pipelining a number of problems of the same kind. For taking out the outputs, immediately after they are produced, a little enhancement of the current SFG is required which has been shown in Fig 4.2.a.

The modified SFG can now be projected onto a linear array which satisfies all the requirements. This is done with $\vec{d}^T = \begin{bmatrix} 0 & 1 \end{bmatrix}$ and $\vec{s}^T = \begin{bmatrix} 1 & 1 \end{bmatrix}$

Hence, $P^T = \begin{bmatrix} 1 & 0 \end{bmatrix}$

i) *Node mapping* :
$$\begin{bmatrix} 1 & 0 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i \end{bmatrix}$$

ii) *Arc mapping* :
$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

iii) *I/O mapping* :
$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 2 \\ j \end{bmatrix} = \begin{bmatrix} 2+j \\ 2 \end{bmatrix}$$

Let the delay element introduced in this step is τ, which is different from the

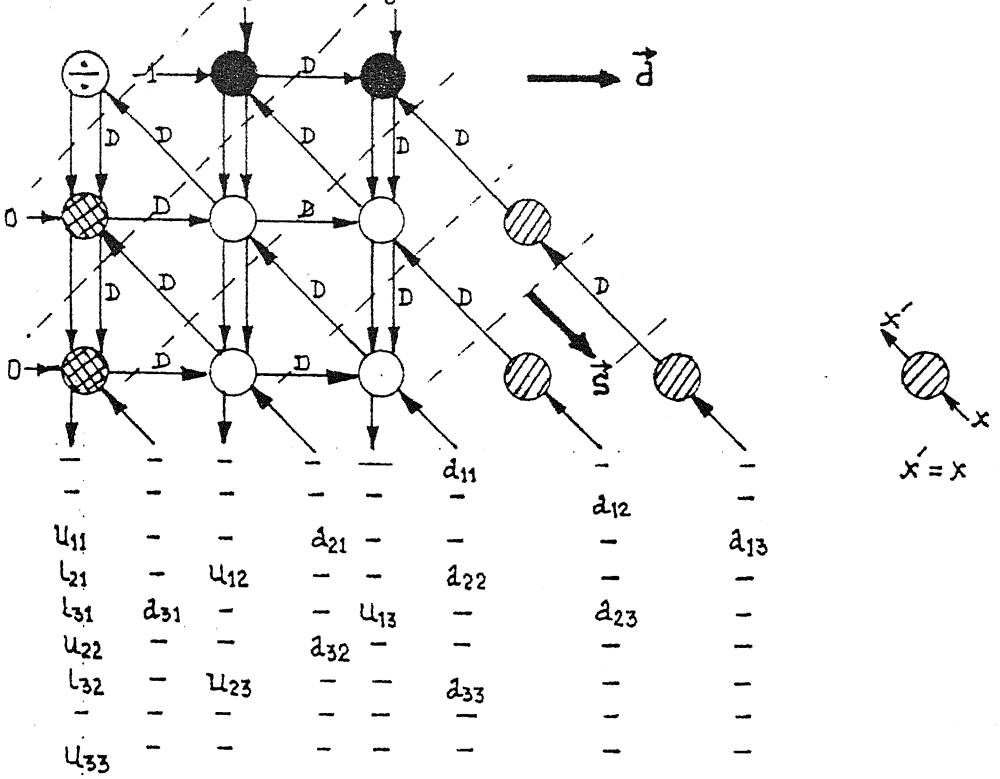


Fig. 4.2 a) Extension of Index Space of the SFG for LU-Decomposition.

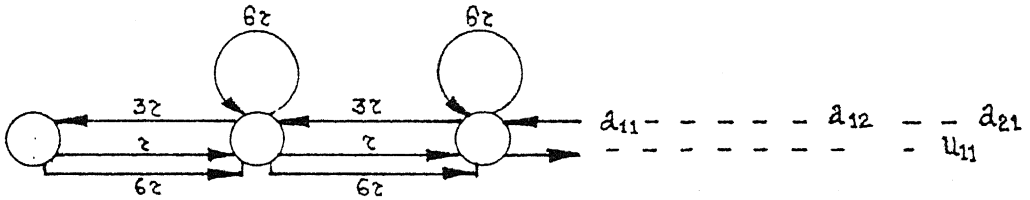


Fig. 4.2 b) SFG After Projection in (01) Direction

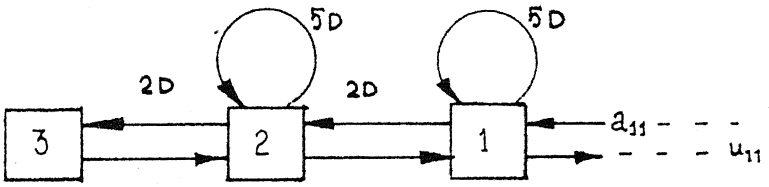


Fig. 4.2.c) Systolic Array Obtained from Fig.(b).

delay D, already present in the SFG.

The relationship between D and τ is governed by the following constraints :

i) *Processor availaility* : For this,

$$D \geq \tau + (N-1)\,(\vec{s}^T \cdot \vec{d})\,\tau$$

Where N is the maximum number of nodes along the $\vec{d}$ direction. This condition guarantees that there is no overlap between two activity-instants which are separated by a delay of D.

ii) *Data availibilty* : For this,

a) $\beta D + (\vec{s}^T \cdot \vec{e})\,\tau \geq 0$ for all edges

b) $\beta D + (\vec{s}^T \cdot \vec{e})\,\tau \geq \tau$ for at least one edge in every cycle.

where $\vec{e}$ is an edge of the SFG and β D is the delay associated with that edge.

condition a) ensures causality of operation.

condition b) ensures that every operation cycle in the new SFG involves at least a single delay.

In the present case $N = 5$ so, $D \geq 5\tau$

We choose $D = 5\tau$

The new SFG is shown in 4.2.b

Step 4. Systolization of the SFG

The systolic array, derived from the modified SFG, is shown in Fig 4.2.c. In order to fit the SFG to the SASP structure, the rightgoing edges had to be multiplexed. Since transmission on these two busses never coincide, multiplexing does not introduce any extra delay. Operation of the cells in each clock cycle has been shown in table 4.1. The operations of the cells involve loops and computation depends upon the indices of the data-items. But they are quite regular, and control flow can easily be controlled through microprogramme residing in each cell.

TABLE 4.1.

	CELL3				CELL2				CELL1			
	OUTPUT		INPUT		OUTPUT		INPUT		OUTPUT		INPUT	
	x'	y'	x	y	x'	y'	x	y	x'	y'	x	y
1.									a_{11}		a_{11}	
2.												
3.												
4					a_{11}		a_{11}					
5.												
6.												
7.		u_{11}	a_{11}						a_{12}		a_{12}	
8.						u_{11}		u_{11}				
9.										u_{11}		u_{11}
10.					a_{12}		a_{12}		a_{21}		a_{21}	
11.												
12.		a_{11}^{-1}										
13.		u_{12}	a_{12}			l_{21}	a_{21}	a_{11}^{-1}	a_{13}		a_{13}	
14.						u_{12}		u_{12}		l_{21}		l_{21}
15.										u_{12}		u_{12}
16.					a_{13}		a_{13}		a_{22}		a_{22}	
17.												
18		u_{12}				a_{11}^{-1}						
19.		u_{13}	a_{13}		a_{22}^{2}		a_{22}	u_{12}		l_{31}	a_{31}	a_{11}^{-1}

Contd.

	CELL3				CELL2				CELL1			
	OUTPUT		INPUT		OUTPUT		INPUT		OUTPUT		INPUT	
	x'	y'	x	y	x'	y'	x	y	x'	y'	x	y
20.						v_{13}		v_{13}				
21										v_{13}		v_{13}
22		v_{22}	a_{22}^{2}						a_{23}		a_{23}	
23						v_{22}		v_{22}				
24		v_{13}						v_{12}		v_{22}		v_{22}
25					a_{23}^{2}		a_{23}	v_{13}	a_{32}^{2}		a_{32}	$y v_{12}$
26												
27		a_{22}^{2-1}										
28		v_{23}	a_{23}^{2}			l_{32}	a_{32}^{2}	a_{22}^{2-1}				
29						v_{23}		v_{23}		l_{32}		l_{32}
30						v_{13}				v_{23}		v_{23}
31									a_{33}^{2}		a_{33}	v_{13}
32												
33		v_{23}										
34					a_{33}^{3}		a_{33}^{2}	v_{23}				
35												
36												
37		v_{33}	a_{33}^{3}									
38						v_{33}		v_{33}				
39										v_{33}		v_{33}

The mapped array can be used to decompose any matrix of band 3, irrespective of its dimension. At a time three matrices can be pipelined, if outputs are taken at the end of execution of the whole algorithm.

§ 4.3 *MAPPING OF QR DECOMPOSITION*

The problem of QR decomposition has recently been given lot of attention after the developement of GIVENS SEQUENCES, which are well suited for parallel computation.

In QR decomposition an m×n matrix A ($m \geq n$) is decomposed into two factors $A = Q \times R$, by applying a sequence of Givens rotations. Q is a m× m orthogonal matrix and $R = \begin{bmatrix} U \\ 0 \end{bmatrix}$ where U is an n×n upper triangular matrix.

The matrix R can be obtained from the equation $R = Q^T A$, where Q^T is a product of rotations, each of which annihilates an element below the main diagonal with the property in which zeros once produced are preserved. The rotation Q (i,j) eliminates the element a (i,j) of A and this process affects only row i and i-1 of A. This is done by the following pre-multiplication.

$$\begin{bmatrix} c_{i,j} & s_{i,j} \\ -s_{i,j} & c_{i,j} \end{bmatrix} * \begin{bmatrix} a_{i-1,1} & a_{i-1,2} & & a_{i-1,n} \\ a_{i,1} & a_{i,2} & & a_{i,n} \end{bmatrix}$$

For updated a_{ij} to be zero,

$$-a_{i-1,j}\, s_{i,j} + a_{i,j}\, c_{i,j} = 0$$

Hence, $c_{i,j} = \dfrac{a_{i-1,j}}{\sqrt{a_{i,j}^2 + a_{i-1,j}^2}}$; $s_{i,j} = \dfrac{a_{i,j}}{\sqrt{a_{i,j}^2 + a_{i-1,j}^2}}$

Thus the process of triangularization consists of steps given below:

For, $j = 1, \ldots\ldots, n$

$$Q_j = \prod_{i=m}^{j+1} Q_{i,j} \quad ; \quad A_j = \prod_{i=1}^{j} Q_i * A$$

where, $Q_{i,j}$ annihilates the i,j th element of the matrix $\prod_{p=m}^{i} Q_{p,j} * A_{j-1}$

Order of annihilation of an 8X5 matrix is given below.

$$\begin{bmatrix} & & & & \\ 7 & & & & \\ 6 & 8 & & & \\ 5 & 7 & 9 & & \\ 4 & 6 & 8 & 10 & \\ 3 & 5 & 7 & 9 & 11 \\ 2 & 4 & 6 & 8 & 10 \\ 1 & 3 & 5 & 7 & 9 \end{bmatrix}$$

Before going through all the steps of mapping procedure, a brief description of the algorithm helps understanding of the problem. For this purpose let's assume one-to-one mapping of matrix elements onto processing elements. The process starts at PE (N-1,1). It fetches $a_{N,1}$ from, PE (N,1) and calculates Givens rotation parameters $c_{N,1}$ and $s_{N,1}$. In the next cycle, these parameters are propagated to PE (N-1,2), then to PE (N-1,3) and so on, while each of the PE's performs (including PE (N-1,1)) the rotation operation in corresponding clock cycle. After PE (N-1,1) finishes the rotation operation, PE (N-2,1) can start its turn to calculate $c_{N-1,1}$ and $s_{N-1,1}$.

The single assignment form of the algorithm for a NXN matrix is given in the next page.

For,

k from 1 to N-1

i from N-1 to K

j from K to N

$$\mathbb{C}x_{i,j}^{k} = \mathbb{M}x_{i,j}^{k-1} \qquad \text{for } i = N-1$$

$$= \mathbb{M}y_{i+1,j}^{k} \qquad \text{for } i \neq N-1$$

$$\mathbb{C}y_{i,j}^{k} = \mathbb{M}y_{i,j}^{k-1}$$

$$\mathbb{C}_{i,k}^{j} = \frac{\mathbb{C}y_{i,j}^{k}}{\sqrt{\mathbb{C}y_{i,j}^{k\,2} + \mathbb{C}x_{i,j}^{k\,2}}} \qquad \text{for } j = k$$

$$= \mathbb{C}_{i,k}^{j-1} \qquad \text{otherwise}$$

$$\mathbb{S}_{i,k}^{j} = \frac{\mathbb{C}x_{i,j}^{k}}{\sqrt{\mathbb{C}y_{i,j}^{k\,2} + \mathbb{C}x_{i,j}^{k\,2}}} \qquad \text{for } j = k$$

$$= \mathbb{S}_{i,k}^{j-1} \qquad \text{otherwise}$$

$$\mathbb{M}y_{i,j}^{k} = \mathbb{C}_{i,k}^{j}\,\mathbb{C}y_{i,j}^{k} + \mathbb{S}_{i,k}^{j}\,\mathbb{C}x_{i,j}^{k}$$

$$\mathbb{M}x_{i,j}^{k} = -\mathbb{S}_{i,k}^{j}\,\mathbb{C}y_{i,j}^{k} + \mathbb{C}_{i,k}^{j}\,\mathbb{C}x_{i,j}^{k}$$

where, $\mathbb{M}y_{i,j}^{0} = a_{i,j}$ for, i = 1 to N-1, j = 1 to N

and, $\mathbb{M}x_{N-1,j}^{0} = a_{N,j}$ for, j = 1 to N

Step 1. DG design

The DG can be easily formed from the single assignment form. This has been shown in Fig 4.3.

Step 2. SFG design

The SFG is being mapped along two projection directions.

1. $\vec{d}^T = [0\ 1\ 1]$; $\vec{s}^T = [0\ 0\ 1]$; $P^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix}$

i) *Node mapping*

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix} * \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i \\ j-k \end{bmatrix}$$

ii) *Arc mapping*

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix}$$

iii) *Input mapping*

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix} * \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ i \\ j-1 \end{bmatrix}$$

iv) *Output mapping*

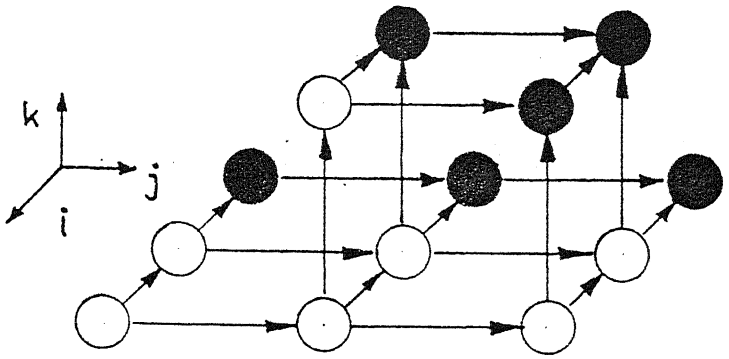
$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix} * \begin{bmatrix} k \\ j \\ k \end{bmatrix} = \begin{bmatrix} k \\ k \\ j-k \end{bmatrix}$$

The corresponding SFG is shown in Fig 4.4.a.

2. $\vec{d}^T = [0\ 0\ 1]$; $\vec{s}^T = [0\ 0\ 1]$; $P^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$

i) *Node mapping*

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix}$$



outputs are produced at darkened nodes.

Fig. 4.3 DG for QR Decomposition

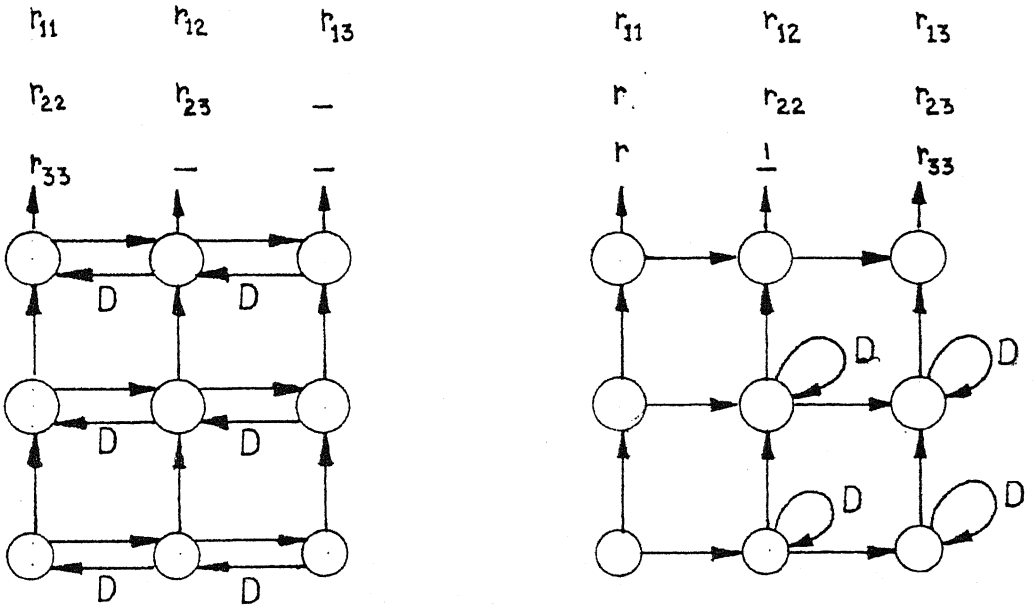


Fig. 4.4. SFG for QR Decomposition by Projection in a) (011)Direction b) (001) Direction.

ii) *Arc mapping*

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

iii) *Input mapping*

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ i \\ j \end{bmatrix}$$

iv) *Output mapping*

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} k \\ j \\ k \end{bmatrix} = \begin{bmatrix} k \\ k \\ j \end{bmatrix}$$

The SFG obtained from this projection is shown in Fig 4.4.b.

Step 3. Projection of SFG

The SFG shown in Fig 4.4.a can be mapped onto a linear array using one more projection along $\vec{d}^T = [0\ 1]$ with $\vec{s}^T = [0\ 1]$. This has been shown in Fig 4.5.a. Preloading of inputs to interior cells can be avoided by modifying the SFG as shown in Fig 4.5.b.

Step 4. Systolization

Finally, from the modified SFG, the systolic array can be easily derived as shown in Fig 4.5.c.

§ 4.4 *EIGEN VALUES OF A SYMMETRIC MATRIX*

The QR decomposition algorithm can be used to find the eigen values of a symmetric matrix. In this method, a matrix A (nXn) is first converted into a tridiagonal form by a series of orthogonal transformation. Then a series of QR

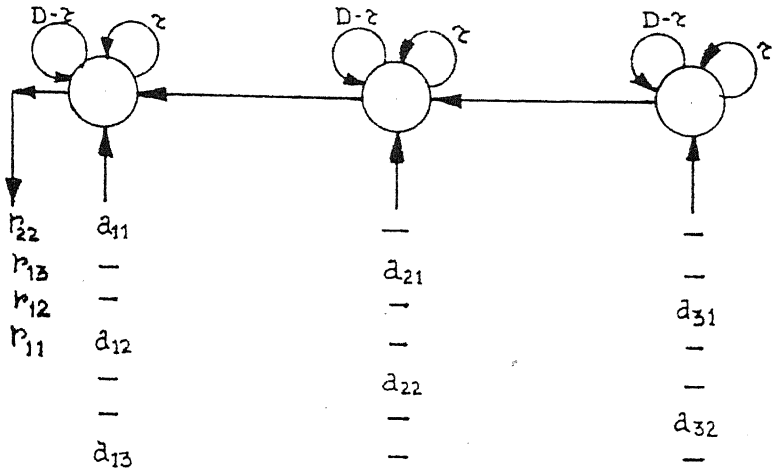


Fig. 4.5 .a) Projection of the SFG in Fig. (4.4 .a) in (01) Direction.

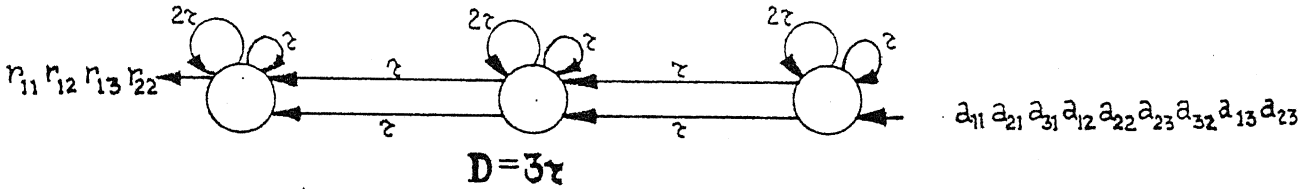


Fig. 4.5 b) Cutset Retiming of the SFG in Fig. a.

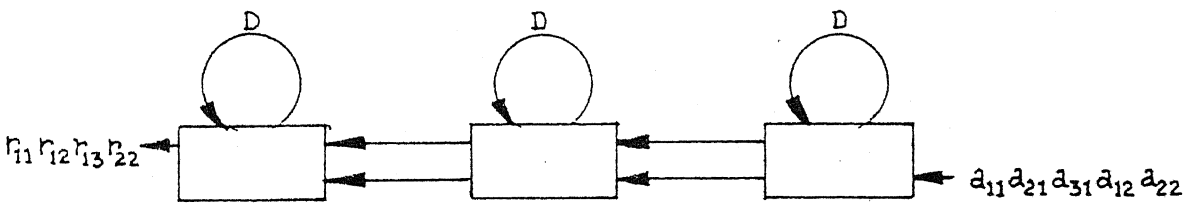


Fig. 4.5 c) Systolic Array Obtained from Fig. b.

algorithms are repeatedly applied to reduce the subdiagonal elements to zero [Alg85]. The QR algorithm is described below,

$$A = Q * R \qquad \text{and} \qquad R = Q^T * A \qquad \text{hence,} \quad R * Q = Q^T * A * Q$$

Thus $R * Q$ is similar to A, where A is a symmetric matrix. Therefore, repeated application of such similarity transforms can be used to find out the eigen values of the original matrix.

In the QR decomposition described in previous section, we carried out only the pre-multiplication by a series of orthonormal matrices $Q_{i,j}$. In addition to this, QR algorithm also requires post-multiplication by $Q_{i,j}^T$. The post-multiplication operation will effect elements of column i and i-1, and since A is symmetric, the (j,i) th element will be annihiliated as a result of this. It is to be noted that for a symmetric matrix, the informations regarding the elements above the main diagonal are redundant and, therefore, can be ignored.

The eigenvalue problem can be done in two steps.

i) *Tridiagonalization*

In this step matrix A is trasformed into a band matrix of bandwidth 2 by applying the QR algorithm. This is described below.

For, $j = 1, \ldots\ldots, n$

$$Q_j = \prod_{i=n}^{j+2} Q_{i,j} \qquad ; \qquad A_j = \prod_{i=1}^{j} Q_i * A * Q_i^T$$

where, $Q_{i,j}$ annihilates the i,j th element of the matrix $\prod_{p=n}^{i} Q_{p,j} * A_{j-1}$

ii) *Reduction of subdiagonal elements*

Once the matrix has been tridiagonalized, the QR algorithm can be repeatedly

applied to reduce the subdiagonal elements to zero. This can be stated mathematically as,

For, $j = 1, \ldots\ldots, n$

$$Q_j = \prod_{i=j+2}^{j+1} Q_{i,j} \quad ; \quad A_j = \prod_{i=1}^{j} Q_i * A * Q_i^T$$

where, $Q_{i,j}$ annihilates the i,j th element of the matrix $\prod_{p=j+2}^{i} Q_{p,j} * A_{j-1}$

The algorithm described above can be efficiently executed on a linear array. In fact, use of a two-dimensional mesh array will prove inefficient, since, the PE's outside the band will be active only during tridiagonaltion, and therefore, remain idle for most of the execution time. The DG and SFG would be almost the same as those designed for QR decomposition, since data dependencies are quite similar in both the cases. A systolic array derived from the SFG shown in Fig 4.4.b is shown in (Fig 4.6). The details of cell operations will not be discussed here.

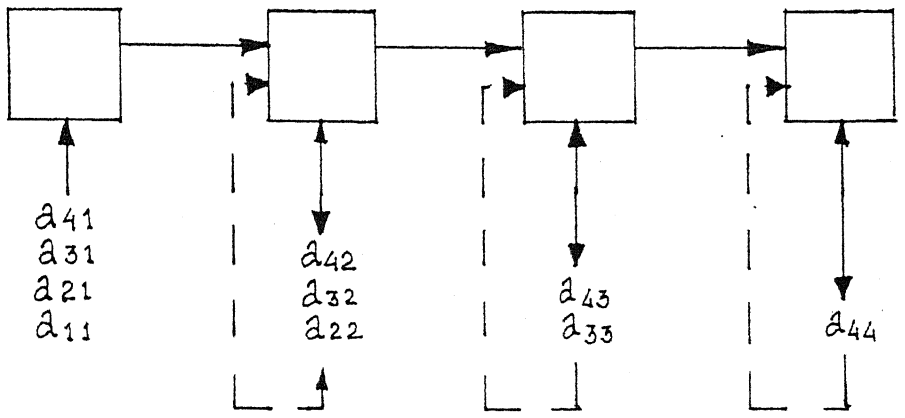


Fig. 4.6 Systolic Array for Determining Eigen Values of a Symmetric Matrix.

Chapter 5

IMPLEMENTATION OF INTERFACE CONTROLLER

§ 5.1 *INTRODUCTION*

In Chapter 2-4 the SASP architecture and mapping of algorithms on this linear array have been described. Now we will discuss about realization of the system on hardware. From the overview of the architecture given in Chapter-2, we see that the SASP array consists of two kinds of basic functional units — interface and cell. Once these two units are developed, the systolic array can be realized by connecting an array of replicated cells to the interface. In the current work, it was decided to build the interface first,— the rationale being that once a proper interface is made, one can have an array of different degrees of versatility using processing elements of varying complexity. A cell can be anything from a simple MAC unit to a sophisticated computational unit depending upon applications, whereas the functions of the interface are more clearly defined and hence it can be made with a design that needs modification less frequently.

In this chapter, the design and implementation of a part of the interface is discussed. The SASP machine is recommended for 32-bit floating

point operations, but for implementing a 32-bit machine, chip count becomes prohibitive considering the PCB-layout facility available here. So the prototype was attempted with 8-bit wide data bus and even then, to accommodate a large number of chips we used wire-wrapping technique instead of going for PCB mounting. Before discussing the design aspects, it is necessary to have a detailed identification of the functions of the interface.

§ 5.2 *FUNCTIONS OF THE INTERFACE*

The systolic array requires input data at a particular rate determined by the algorithm to be executed, and it gives the output data at a rate which is also specific to the problem. During execution of large size problems on a fixed small size array, intermediate results are generated, which have to be fed back again into the processor-array until the final output is available.

All these data handling operations cannot be done by the host with which the systolic array is used as an attached processor. The reason is: the supply rate and reception rate of data have to match the speed of operation of the array. Furthermore, it is more desirable to keep the array operation totally transparent to the host operation.

Moreover, as shown in the SASP architecture (Fig. 2.1), the adr-bus is used to pump address patterns through the cells. The address sequence has to be generated at the interface. Thus the interface should act as a competent source for all interconnection channels (X-bus, Y-bus, adr-bus), as well as a suitable sink for intermediate results and output.

There is one more function which the interface is required to perform – loading of data and microprogram into each cell. As discussed in Chapter-2, all the cells are connected to a global BC-bus, which is used to dump microprogram and resident data in respective cells. The control of this BC-bus rests with the interface. A list of jobs for the interface is given below.

- ***Host-state***

i) Receiving data from the host and sending output to the host.

ii) To direct proper sequence of controls for loading data and microcode into individual cells.

iii) To take over control of the array from the host before execution starts.

- ***Array-state***

i) To supply X and Y data to the array at required rate.

ii) To route data according to the configuration of the array (forward or reverse).

iii) Receiving intermediate results and looping them back into the array.

iv) Receiving and storing output results.

v) Generation of address for the adr-bus.

vi) Returning control back to the host through an interrupt when execution is over.

§ 5.3 *BLOCK DIAGRAM DESCRIPTION OF THE INTERFACE*

The block diagram of the interface is shown in Fig 5.1

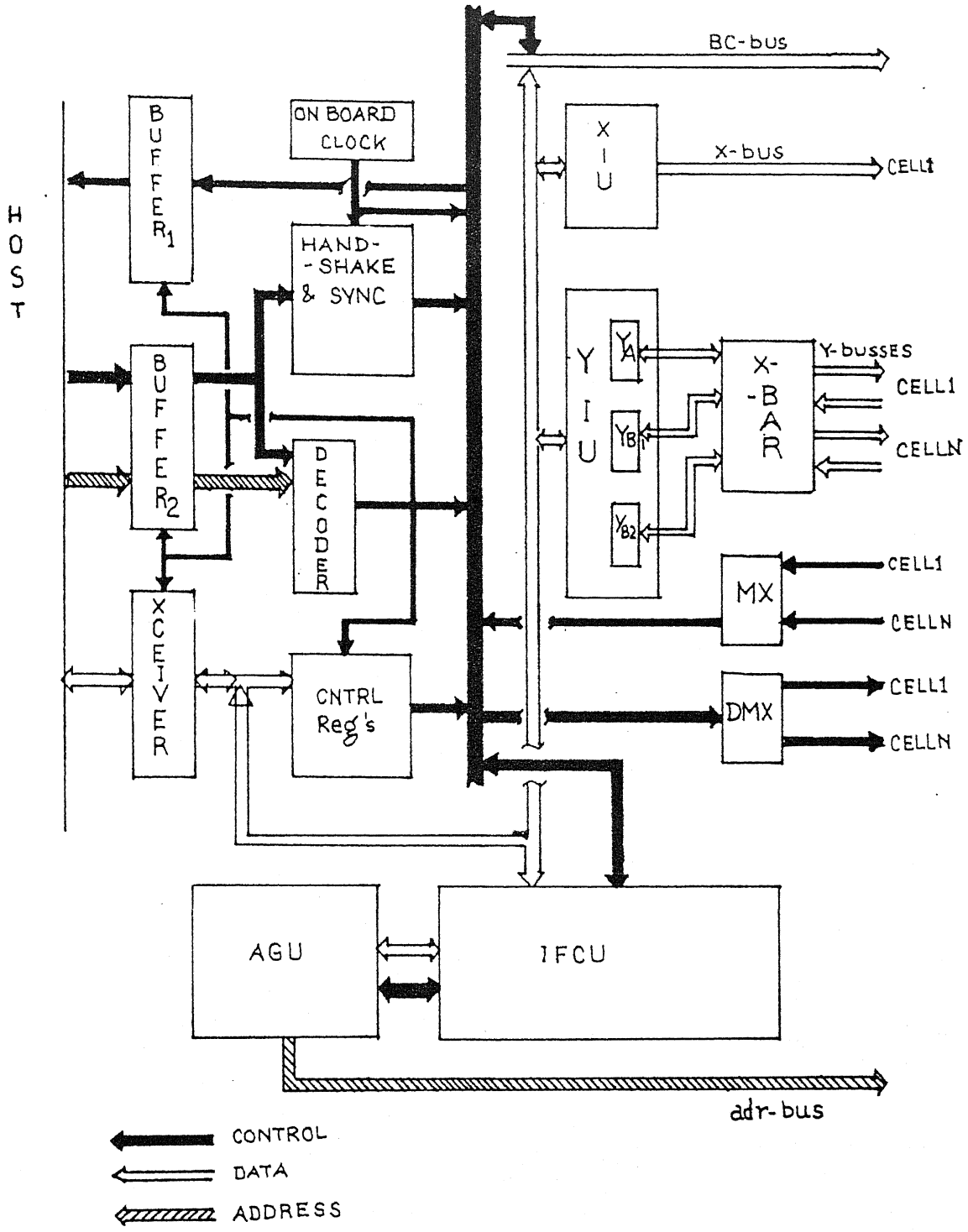


Fig. 5.1 Block Diagram of the Interface.

The interface can be functionally divided into four functional units.

1. *Host interface unit* (BUFFER1, BUFFER2, XCEIVER, DECODER)

The host interface unit buffers the address, data, and control lines coming from the host. The decoder selects proper blocks (e.g. CNTRL REG's, XIU, YIU) for writing or reading of data. The host considered here, is a PC–XT and SASP is mapped on its I/O address space. The control lines going to the host are *Ready* and *Interrupt* signals.

2. *Data interface unit* (XIU, YIU, X-bar, MX, DMX)

Before asking the systolic array to start operation, the host dumps the data onto the RAMs of the data interface unit, wherefrom these data are transmitted from cell to cell. XIU contains X-data which, during operation, is fed to cell1. YIU sends Y-data from RAM Y_A, stores output results in RAM Y_A; and receives and recirculates intermediate results through RAM Y_{B1} and RAM Y_{B2}. For handling intermediate results, two RAMs have been used, so that simultaneous read and write can be supported by accessing both of them at the same time (i.e. writing into one RAM and reading from the other). X-bar provides appropriate routing of data channels for forward and reverse mode of operation (§ 2.3), and MX, DMX are used to send/receive control signals to/from appropriate cell in these modes.

3. *Host side control* (HANDSHAKE & SYNC, CNTRL REG's)

The host controls the interface operations during data and microprogram loading through setting some bit of the CNTRL REG's, which are

mapped as output ports on the host address space. Data transfer between host and interface is done by handshaking. As the PC initiates a read or write cycle meant for the interface, the *Ready* line goes low, the **HANDSHAKE & SYNC** unit generates local read or write pulse, and after completion of the required operation, *Ready* line is asserted to indicate the end of operation.

4. ***Array side control and address generation unit*** **(IFCU, AGU)**

The **IFCU** takes over control of all the blocks after host has finished loading of microcode and data, and supervises all kinds of routing and exchange of data during execution of an algorithm. After the execution is finished, it generates an interrupt which is passed to the host through the host interface unit, asking the host to take necessary action. The **AGU** section generates the address pattern for the adr-bus.

Although access of data by processor array generally occurs at regular intervals, the regularity can be disturbed for large-size problems where the fixed (small) size of the array requires partitioning of the main problem. This indicates that tracking of instants of data access, by event counting through up/down counters, lacks generality. For this reason it was decided to put the interface under microcode control which increases the versatility of the unit.

The **AGU** is functionally equivalent to a self contained ALU and is capable of generating complicated data dependent address patterns under the control of a rich instruction set. The instructions come from the microprogram, and thus the process of address generation has a close coupling with microcode

sequencing.

In the following section, implementation of the IFCU and AGU, as done in this project, will be discussed.

§ 5.4 *DESIGNING 'IFCU' & 'AGU'*

§ 5.4.1 *INTERFACE CONTROL UNIT* (IFCU)

The IFCU has been implemented as a programmable microengine. At this point it would be relevant to discuss the principle and some of the advantages of microcode control in a nutshell.

It is a standard practice to use microprocessors to control data flow and to perform computations in a 'smart' circuit. But the inherent sequential nature of microprocessor operation prevents its use in high throughput machines, where more functional parallelism is required.

This parallelism can be achieved by using microcode approach. The main difference between microprocessor circuits and microcode circuits is that the functional units integrated in a microprocessor are spread out as separate building blocks in a microcoded circuit, so that they can operate simultaneously and independently.

In order to coordinate the independently operating devices, these functional blocks are operated in tandem in synchronism with a common system clock. Control instructions for all the devices are put together in a central microcode memory. Instructions are stored in each location of the memory, and a

memory location is accessed in every system clock cycle. The instruction can be group of bits for VLSI devices or can be single bit control signals for SSI or MSI devices.

In a high performance system, mere sequential addressing of microcodes is not sufficient. It is desirable to have a sophisticated program flow, that accommodates nested loops, subroutines, interrupts etc. Such demanding sequencing requirements are met by using a program sequencer. Like other functional units, the sequencer also gets its instructions from the microcode memory and generates the address for the next instruction depending upon the current instruction.

ADSP-1401 program sequencer has been used as the microprogram controller. The preliminary block diagram for the IFCU and the AGU has been shown below (Fig-5.2).

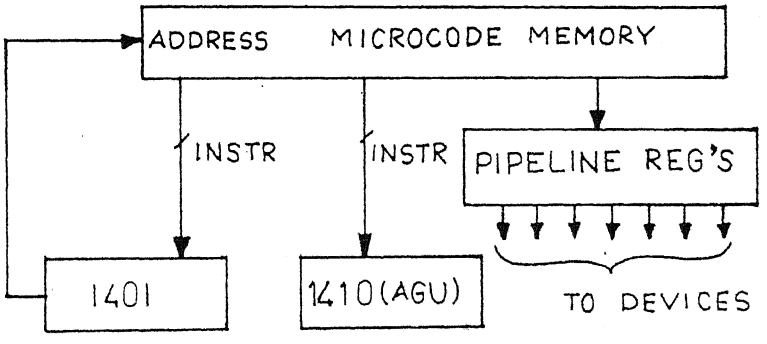


FIG. 5.2 Preliminary block diagram of 'IFCU' & 'AGU'

The pipeline register is used to latch the control signals for the functional blocks. Since the microcode memory is accessed in each system clock cycle, data from the memory is unstable for some time during memory access. This can cause uncerteinty in the operation of the devices controlled by the microcode. Pipeline register prevents this by latching data only when it is stable. This function will be clear from the following timing diagram (Fig 5.3).

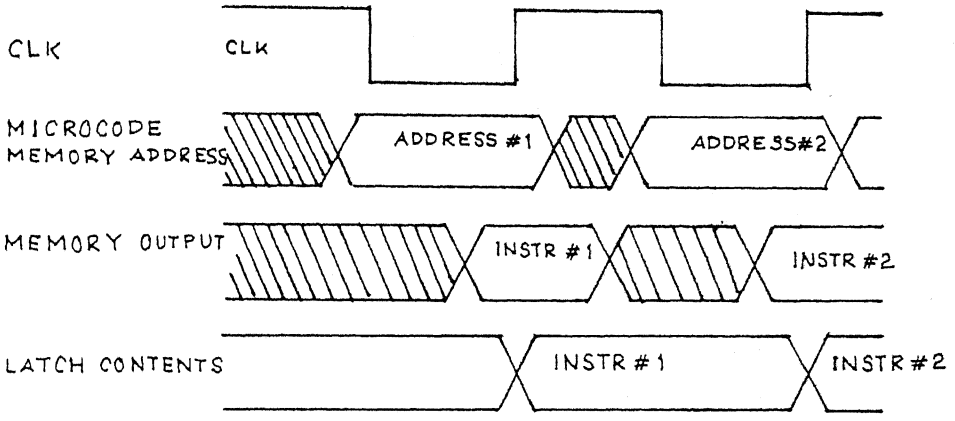


FIG. 5.3 Pipeline latch timing

§ 5.4.2 *ADDRESS GENERATION UNIT (AGU)*

The interface has to generate the address on the adr-bus which flows systolically from cell to cell. The ADSP-1410 address generator has been used for flexible address generation. The ADSP-1410's architecture features a 16-bit ALU, a comparator, and 30 16-bit registers. In a single cycle the device can output a 16-bit address, modify this address, and can do conditional loop back to the top of a circular buffer. Consequently it supports modulo addressing without any overhead. 1410 can do bit reversal at the output, which is very important for FFT algorithms.

§ 5.4.3 *DOWNLOADING OF MICROCODE*

Generally, microprogram of a system is burned into a ROM. But in the present case, function of the IFCU is determined by the algorithm to be executed and the microcode for the control unit changes with different algorithms. Hence, a flexible controller needs to be reconfigured dynamically. This can be achieved by downloading the microprogram from the host, before starting operation.

ADSP-1401 has a special instruction (WCS) to support this operation. In this configuration (Fig 5.4), microprograms are loaded in RAMs and additional circuitry is added to support download.

WCS *instruction* :

The WCS instruction puts the sequencer in a mode in which sequential addresses are output every cycle and instructions are ignored. The *Flag* input is used for handshaking. When the WCS instruction (20 hex) is presented to the instruction port of the sequencer with the starting address (for microcode RAM) presented

to the data port, the instruction (20 Hex) is latched with the rising edge of the next clock cycle and subsequent instructions are ignored.

After receiving this instruction, the program sequencer outputs the starting address presented at the data port and waits for the *Flag* input to be high. Once the *Flag* is high, the sequencer increments the address and outputs that in the next clock cycle. The *Flag input* is used for handshaking and can be made high permanently for fast devices.

The download mode can be terminated through an interrupt or reset. In present circuit, termination has been done by external reset.

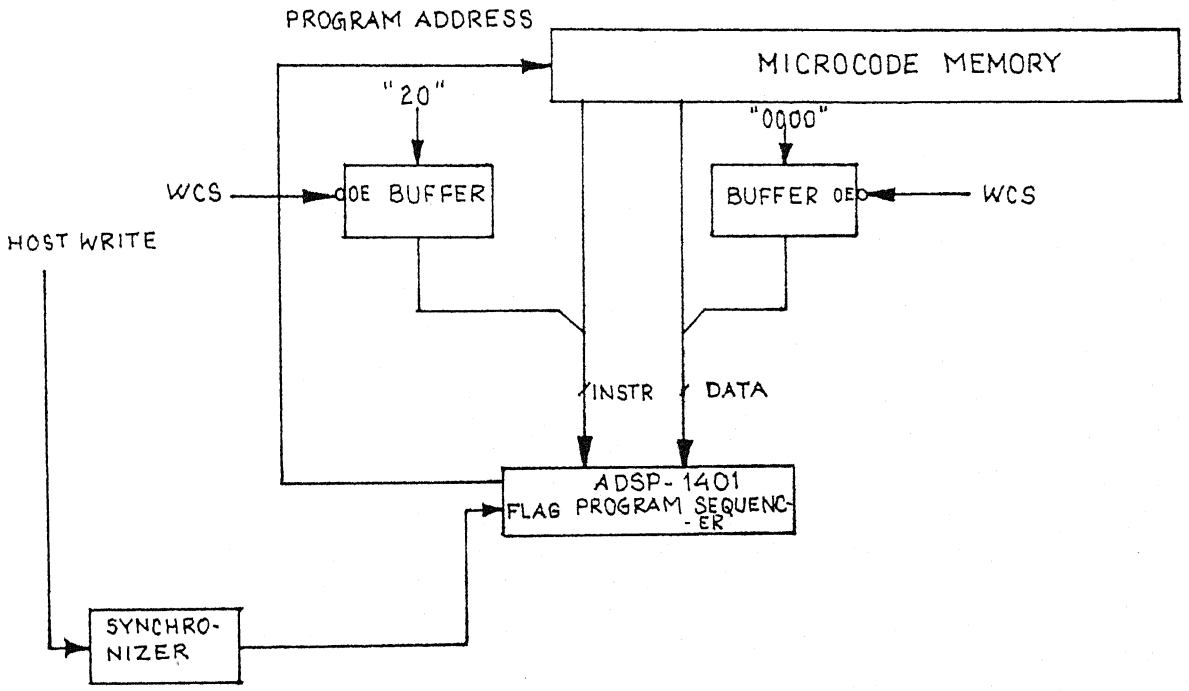


FIG. 5.4 Circuit for downloading microprogram

In Fig 5.4, the WCS instruction and the starting address are given through hardware. For this, a control bit called *Wcs* is made logic low and the buffers containing the WCS instruction and starting address are enabled, and they are put on appropriate buses. With this the sequencer goes into WCS mode. The write signal from the host is synchronized to generate the *Flag* input.

After the downloading has been finished, the host resets the sequencer, clearing WCS and causing a jump to location 0 (the first location of the microprogram).

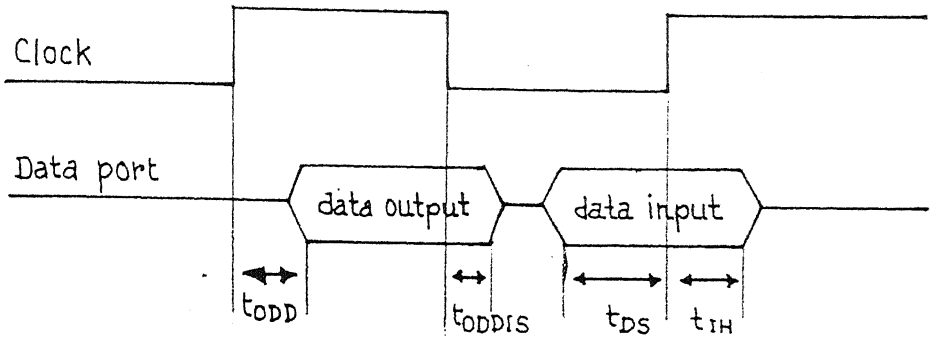
§ 5.4.4 *DATA TRANSFER BETWEEN 'IFCU' & 'AGU'*

The ADSP-1401 and ADSP-1410 has a 16-bit data port each. Allocation of two different microcode fields to these ports would have made the arrangement most flexible, but it would cause inefficient utilization of the memory space.

A reasonably flexible scheme for data transfer among microcode memory, address generator and program sequencer is given in Fig 5.5.b. The output and input arrangement of ADSP-1401 and 1410 permits data to be output during clock phase one, while inputting of data is performed in clock phase2, thus allowing reading and writing of data in single clock cycle (Fig. 5.5.a).

The circuit of Fig 5.5.b allows following data transfers in a single clock cycle.

- DF→ 1401 or DF→ 1410 (*Ken* = 1, *Den* = *Dstb* = 0) : Required for initialization of 1401 & 1410 registers. Supplies 1401 with direct, indirect &



	MIN	MAX	UNITS
t_{ODD}		50	ns.
t_{ODDIS}		20	ns.
t_{DS}	10		ns.
t_{IH}	3		ns.

Fig. 5.5 a) Timing of Data Transfer Through Data Ports of 1401 and 1410.

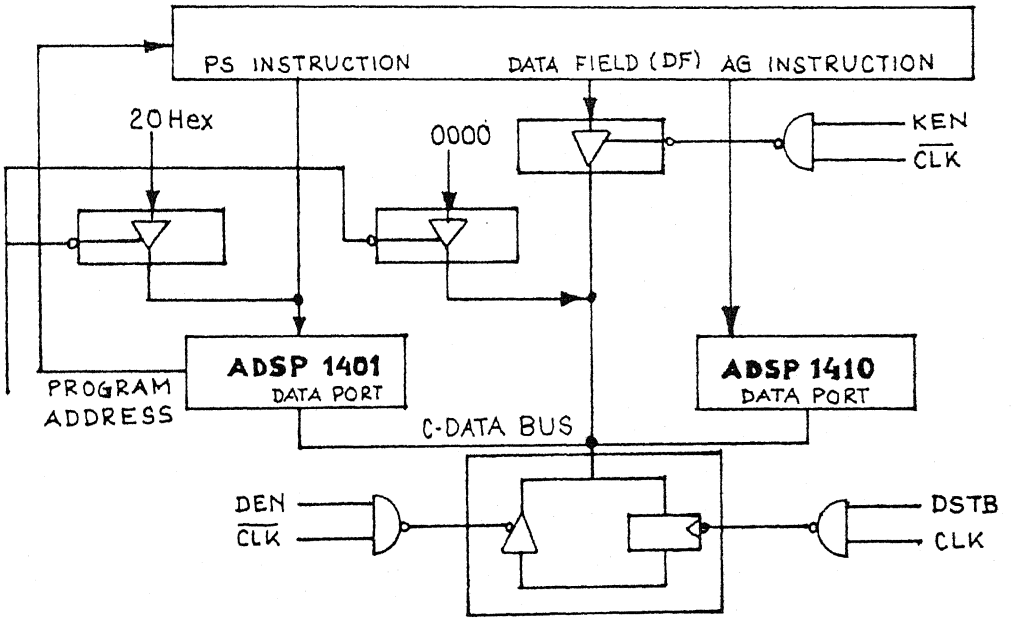


Fig. 5.5.b) Scheme for Data Transfer between IFCU and AGU.

relative program addresses, and 1410 with immediate data addresses or offsets.

• **1410 ↔ 1410** (*Ken* = 0, *Den* = *Dstb* = 1) : It allows 1410 to save its registers in 1410's stack during a subroutine call, and allows 1401 to use the 1410 as an ALU to calculate program address.

Timing diagram of these data transfers are shown in Fig 5.6.

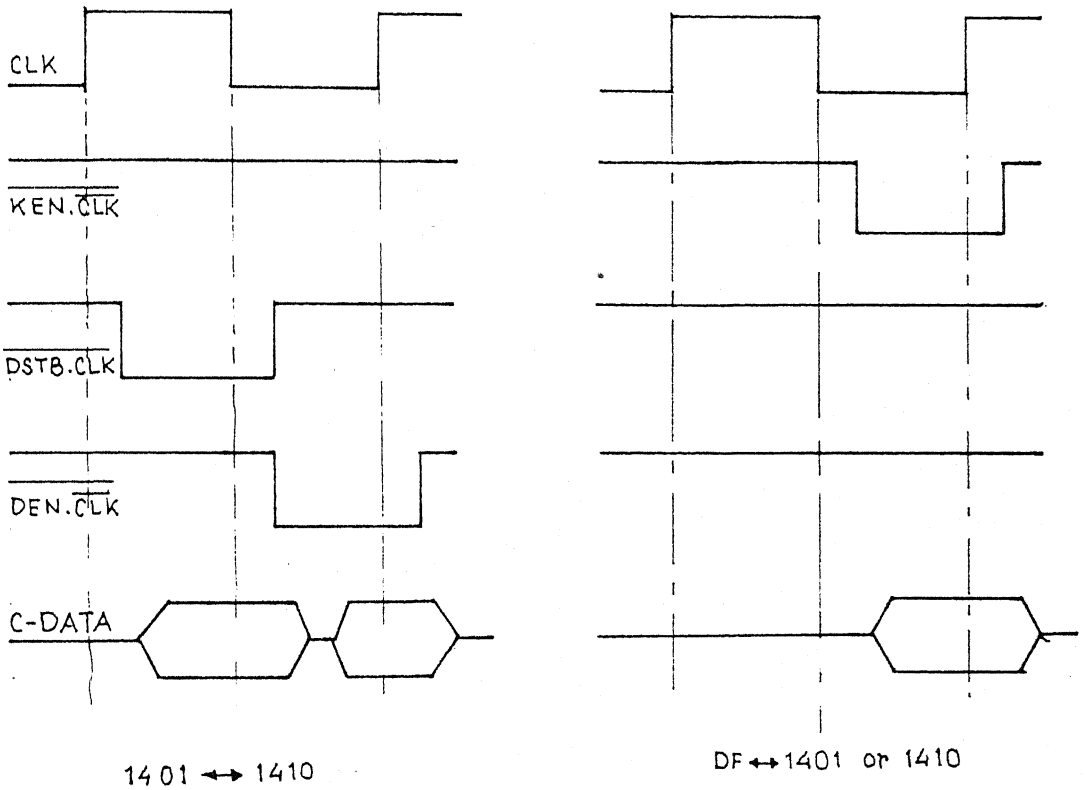


FIG. 5.6 Data transfer timing

§ 5.5 *CIRCUIT DESCRIPTION*

It has already been mentioned in Chapter-2, that we treat the interface as a special cell (cell$_0$). In line with this configuration, the **IFCU** gets its signals from the BC-bus (§ 2.5) when the cell address (BCA_0–BCA_3) matches the address of cell$_0$.

IFCU has eight 8kX8 RAMs for storing microprogram and therefore, each microcode word is 64-bit wide. The host controls the states of the control unit through a number of control bits which come from a set of control registers (**CNTRL REG's**). These control bits are listed below (a brief outline of their functions has already been given in Chapter-2; more details will be given along with the circuit description.):

1. Lμc 2. Sync 3. $\overline{\text{Clr}}$ 4. Rst 5. $\overline{\text{Wcs}}$

6. Host/ $\overline{\text{array}}$ 7. BCA_0-BCA_3

Furthermore, for writing and reading (microcodes need not be read as they are loaded by the host only. But this option has been kept to facilitate testing) of microcode, **IFCU** uses the following signals from the **Handshake & SYNC** block.

1. $\overline{\text{Wr}\mu\text{c}}$ 2. $\overline{\text{Rd}\mu\text{c}}$ 3. Flg

Circuit diagram and timing for these signals are given in Fig. 5.7. Before going through it, it is necessary to mention two points about data interface. Firstly, since the **Handshake & SYNC** unit generates local read and write signals, the data from **IFCU** to host is latched with a signal called *SYIOR* (Fig. 5.7). Secondly, all data transfers on the BC-bus are treated by the host as if it is being done with a single I/O port i.e. when the host addresses a particular port (i.e. $\overline{\text{Cs}\mu\text{ch}}$=0), only then the write and read signals appear on

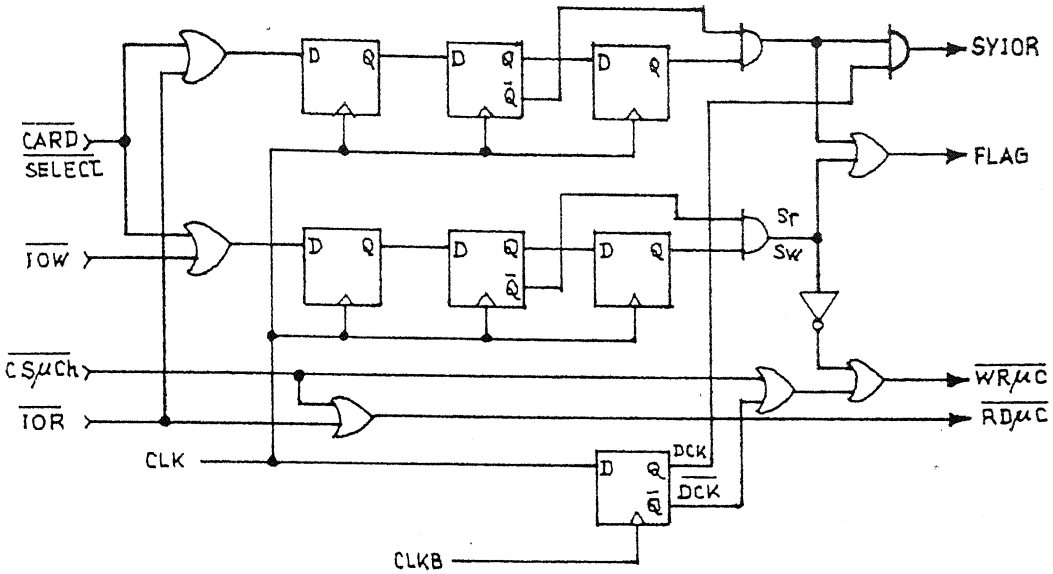


Fig. 5.7.a.

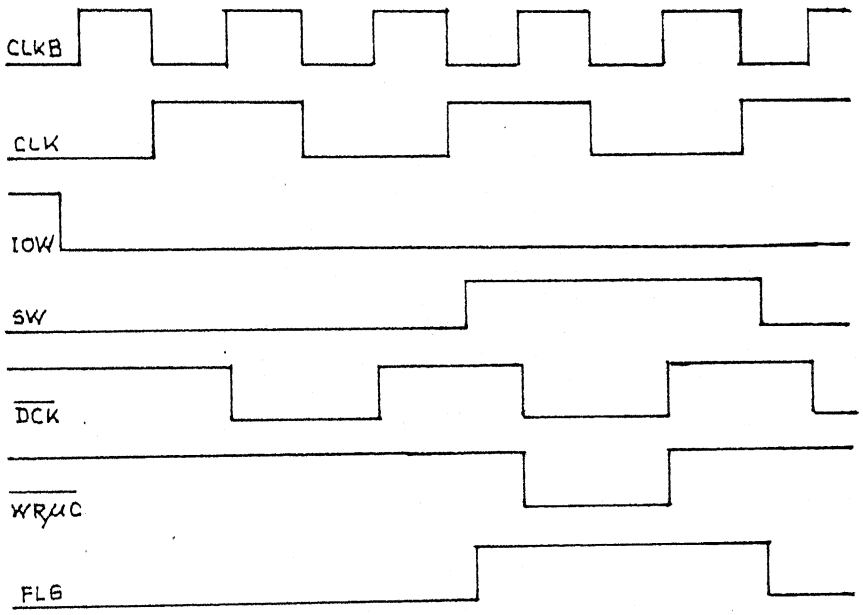


Fig. 5.7.b.

Fig. 5.7 Generation of Handshake for Reading/Writing Microcode and Data.

a) Circuit b) Timing.

the BC-bus.

§ 5.5.1 *MICROCODE LOADING*

Schematic diagram of the microcode loading scheme is given in Fig. 5.8. Most of the control signals come through a transparent latch (**LTH1**), while the remaining control signals and data are adequately buffered (**BF1, BF2**). The **BUFF-Bank** consists of eight bidirectional buffers. Each of them are connected to one RAM of the **RAM-Bank** (it is an array of eight RAMs labeled as $\mu R_0 - \mu R_7$). A counter and decoder combination selects one pair of RAM and buffer at a time while loading the microcode. But during microprogram execution, all the buffers are disabled and all the RAMs are enabled. It is to be noted that *Lwt* = 1 for microcode loading and = 0 for microcode execution.

The microcode is loaded in a column major order. As mentioned before, the microcode reside in an array of 8 bytes of columns and 8k bytes of rows. Each RAM holds a column of microprogram and at a time one column is loaded. Loading starts with with μR_0 and after the writing is finished, the sequencer is reset only to start another sequence of WCS cycles to write onto the next RAM. Each time the counter (CTR) is incremented to select the proper RAM.
In the following part, the sequence of operations that occur while loading a microprogram, is listed.

1. **At power up :**

   - *F=1 ;* Output of **LTH1** is tri-stated. $\overline{Rd\mu c}$ & $\overline{Wr\mu c}$ are pulled high. F is set to zero by a software command.

2. **Loading of microprogram :**

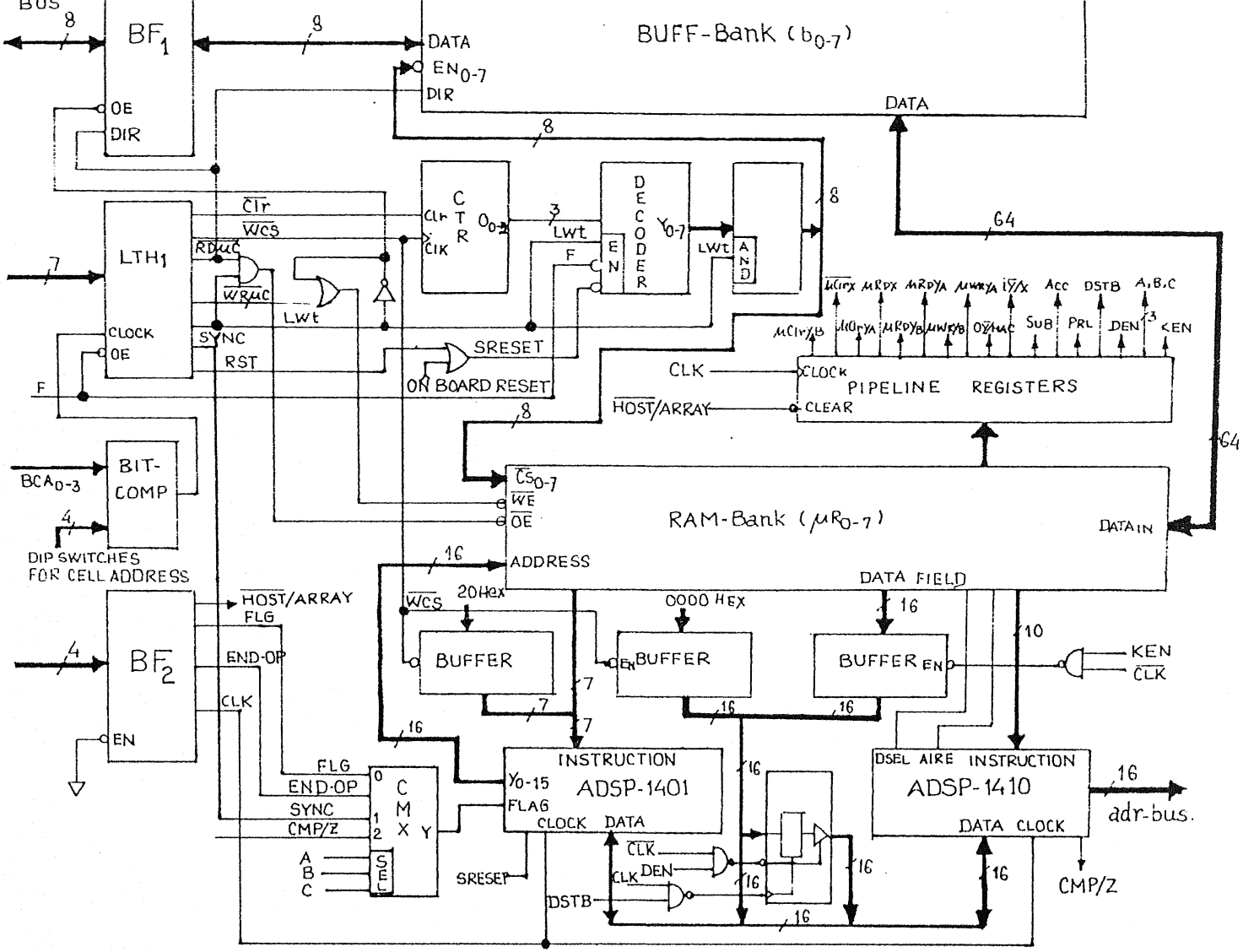

BUS
BF1
DATA
EN0-7
DIR
OE
BUFF-Bank (b0-7)
DATA
CTR
DECODER
AND
LWT
F
LTH1
CLOCK
SYNC
RST
WCS
RDμC
WRμC
SRESET
ON BOARD RESET
CLK
HOST/ARRAY
PIPELINE REGISTERS
CLEAR
BIT-COMP
BCA0-3
DIP SWITCHES FOR CELL ADDRESS
RAM-Bank (μR0-7)
DATA IN
ADDRESS
DATA FIELD
20Hex
0000 HEX
BUFFER
BF2
HOST/ARRAY FLG
END-OP
CLK
EN
FLG
SYNC
CMP/Z
CMX
INSTRUCTION ADSP-1401
FLAG
CLOCK
DATA
DSEL AIRE INSTRUCTION ADSP-1410
DATA CLOCK
adr-bus.
CMP/Z
SRESET
DSTB
DEN
KEN

FIG 5.8

starting address (0000) to the sequencer, remove reset.

- $L\mu c = \overline{Wcs} = 1$; $Rst = \overline{Clr} = 0$; $\overline{Wcs}$ is removed and the sequencer waits for the *Flag* input.
- $L\mu c = \overline{Clr} = \overline{Wcs} = 1$; $Rst = 0$; Clear to CTR is removed and the host can go on writing onto μR_0.

After loading of μR_0 is finished, writing should start at μR_1. The required steps for this are listed below.

- $L\mu c = \overline{Clr} = Rst = 1$; $\overline{Wcs} = 0$; Reset the sequencer.
- $L\mu c = \overline{Clr} = 1$; $Rst = \overline{Wcs} = 0$; Give WCS instruction.
- $L\mu c = \overline{Clr} = \overline{Wcs} = 1$; $Rst = 0$; Wcs is removed, CTR is incremented and hence $\mu R1$ is selected.

Thus, switching from one RAM to other is done and microcodes are written into each of them.

§ 5.5.2 *EXECUTION OF MICROCODE*

Once the microprogram has been loaded, the host directs the IFCU to start execution. This can be done by the following sequence after the last instruction in μR_7 has been written.

- $L\mu c = 0$; $\overline{Clr} = \overline{Wcs} = Rst = 1$; Reset the sequencer. Enable all the RAMs and disable all the buffers in BUFF-Bank. All the RAMS are permanently set to read mode.
- $L\mu c = Rst = 0$; $Sync = \overline{Clr} = \overline{Wcs} = 1$; Remove the reset and start execution.

Note that, before doing the last step, Host/$\overline{arrag}$ must be made logic low. This blocks host control signals to all data control blocks, and connects microcode instruction bits to those units (e.g. XIU, YIU).

According to systolic principle, all the cells should act in synchronism. To ensure this, execution of microcodes in all the cells should start at the same clock. This is done in the following manner. After loading microprogram in a cell (including $cell_0$), the host puts that cell in microcode execution mode. According to the ADSP-1401 specification, the first instruction of any microcode has to be a simple CONT (continue). Now, in every microprogram, a conditional jump statement is deliberately given as the second instruction, which keeps the sequencer waiting for an external condition. The condition nothing but the *Sync* bit from the CNTRL Reg's (Fig 5.1). It is fed to the *Flag* input through the CMx, and is kept low by the host until all the operations to be done in the host-state finishes. Once all the cells are ready with necessary data and microcode, the host makes the *Sync* bit set to logic 1. In the next clock cycle all the cells which had been waiting so long for the external condition starts execution in unison.

§ 5.5.3 *USE OF MICROCODE BITS*

The microcode bits are used to control different blocks as shown in Fig. 5.1. The main functions performed through microcode control are as follows. (Signals written in the brackets are used for the corresponding function.) :

1. **Sending data to next cell (μRdx, μRdy_a, μRdy_b)**

The main function of the microcode is to send X or Y data to appropriate cell according to the array configuration. It has to send a write signal along with the data, to be written into the input queue of the destination cell. The reading and writing of data should be over in a single clock cycle. This can be done by latching the read data as shown in Fig 5.9. The read signal (e.g. μRdx) is

directly sent to the destination cell which generates its write signal by gating the read signal with its own clock.

2. **Setting the configuration of the array**

As mentioned earlier, the **SASP** can operate in two configurations – forward and backward. The difference between them lies in the connection of Y-bus only and switching between these two configurations can easily be done by properly adjusting the **X-bar**, **Mx** and **DMx**. Control bits for these units also comes from the microcode and hence the configuration can be changed dynamically during program execution.

3. **Reporting the end of execution.**

Once the interface has sent all the data required for execution, it anticipates the end of computation and goes on checking the *End-op* condition. Since the last valid computation can be done by any cell, depending upon the size and type of the problem, *End-op* is made an open collector input which is connected to all the cells. The cell, last to finish computation, raises the line to logic one, and after detecting that the control unit sends an interrupt to the host.

In the present work, we have tried to apply microcode control in the above mentioned cases. The versatility of the interface can be enhanced by putting many new functions on it. It would result in an increase in number of functional blocks with very little control overhead. This is because, in a microcode circuit, extra control can easily be provided by increasing the microcode width.

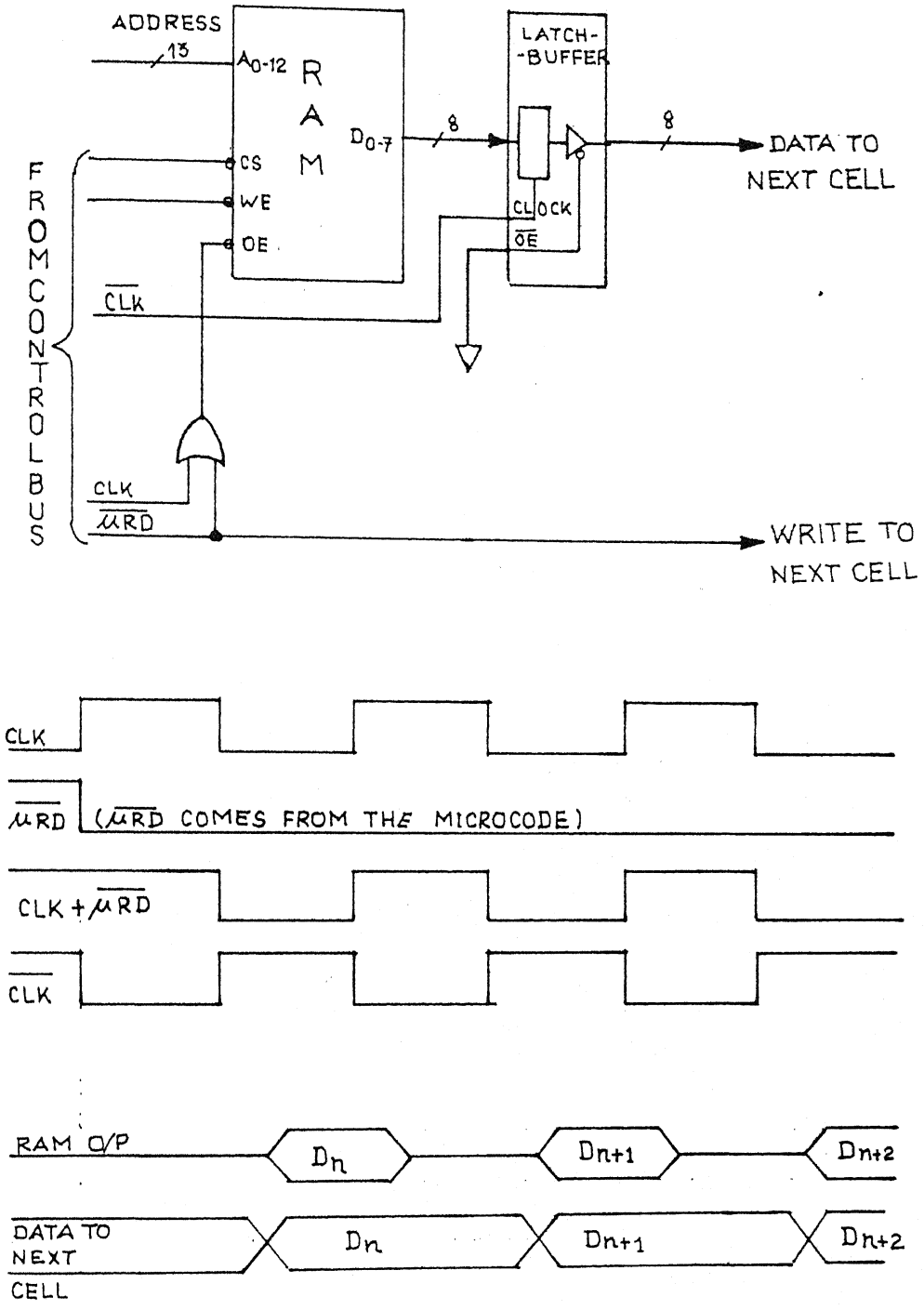


Fig. 5.9 Data Transfer from Interface to Next Cell.

Chapter 6

CELL FOR 'SASP'

§ 6.1 *INTRODUCTION*

Systolic arrays are characterized by rhythmic data flow through an array of processing elements which have a high degree of structural and functional similarity. The power of systolic computation arises from regular and systematic use and transfer of data by various processing elements (PE's) called 'cell's. In the systolic array discussed here, each cell has two data channels both at the input and the output, and an address channel as shown below (Fig. 6.1).

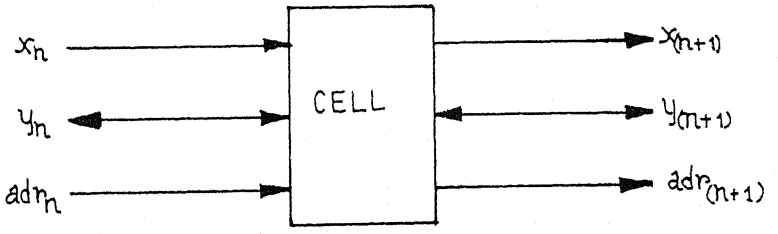


FIG. 6.1 A Systolic Cell

The computation performed by each cell is of the form . . . . . .

$$y_{(n+1)} = w_n * x_n + y_n \ ,$$

$$x_{(n+1)} = x_n \ , \qquad \ldots\ldots\ldots\ldots \quad 6.1$$

where w_k comes from the local memory specified by the address on the adr-bus. This simple cell configuration supports implementation of many useful algorithms, but to get a versatile systolic array computational capacity of the cell must be enhanced. In this chapter, a highly flexible cell architecture will be suggested (the of the cell design can be found in [Us89]), and the implementation of a simple cell will be discussed.

§ 6.2 *CELL FOR 'SASP'*

The data path of the cell is shown in Fig. 6.2. The cell is controlled by a horizontal microengine and all the datapaths are 32-bit wide, which can support 32-bit floating point operations. The cell consists of a queue for each interconnection channel (XQ, YQ & AQ), two Muxes for X and Y channels (XMx, YMx), two memory banks for resident and temporary data (M1, M2), a 32-bit floating point computing unit (ALU, MUL), a register file pair for the floating point units (RALU RMUL), and a local address generator (LAGU). All internal datapaths are connected through a completely general crossbar.

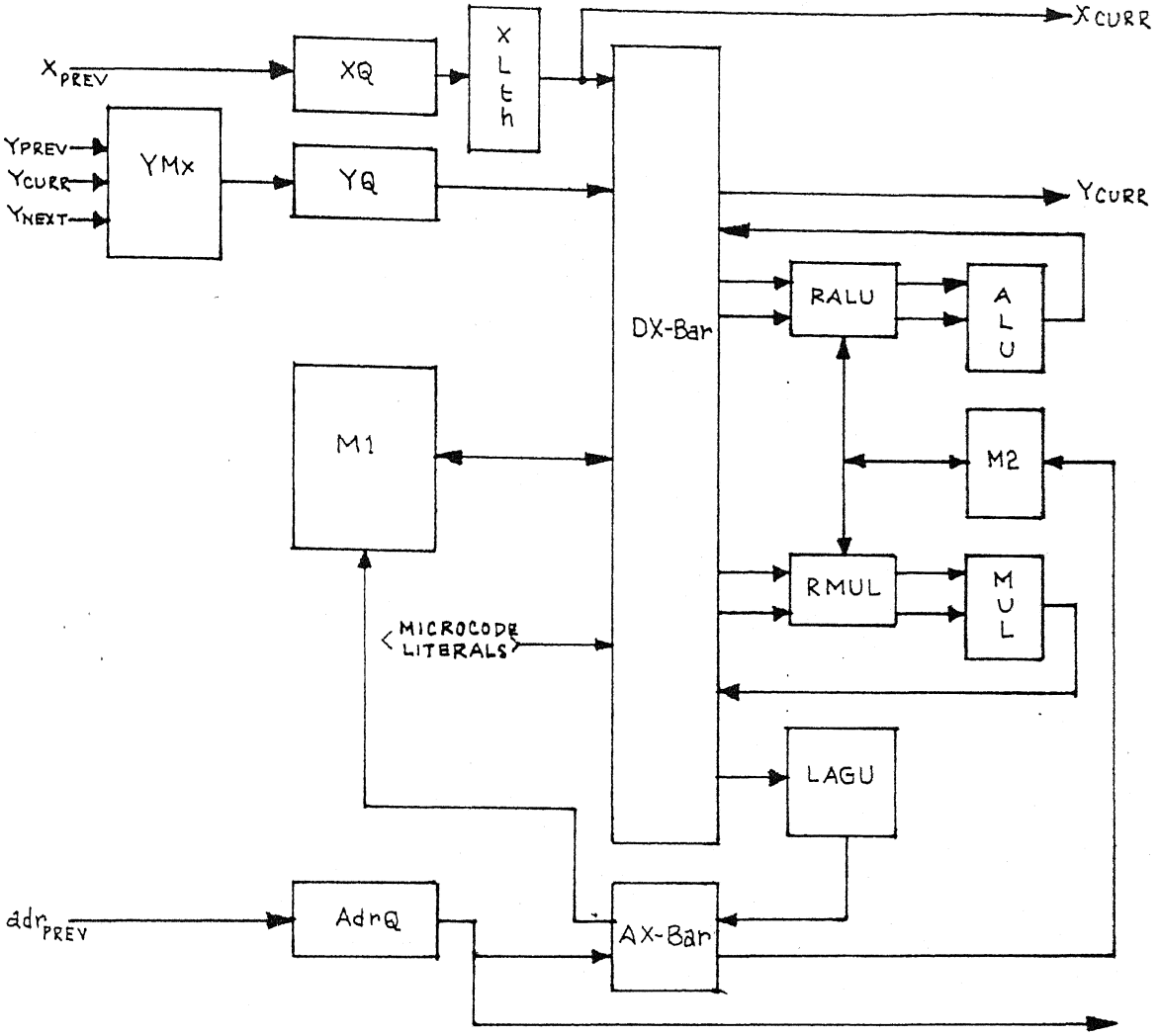


FIG. 6.2 **Block diagram of the cell**

Following is a description of the major blocks of the cell.

- ***Arithmetic Units*** **(ALU, MUL)**

They can be implemented with ADSP-3201 and ADSP-3202 32-bit floating point

chipsets. These chips have one level of pipelining with minimum cycle time of 40 ns and a minimum latency of two clock cycles.

• ***Data storage units*** **(M1, M2, RALU, RMUL)**

The local data memory M1 is used for storing constants, intermediate data, etc. and thereby reducing I/O bandwidth requirement of each cell. By using a local memory for storing intermediate results, a cell can be multiplexed to implement the function of multiple cells and the linear array can be used to implement algorithms designed for two dimensional systolic arrays. The backup memory M2 increases the memory bandwidth. Addresses for both the memories come from the address crossbar. Multiport 64X32 register files (RALU, RMUL) are used for buffering data for ALU and MUL. These data storage components increase the computational bandwidth by permitting flexible data routing through their five ports. ADSP 3128 register files can be used for this purpose.

• ***Crossbar*** **(AX-Bar, DX-Bar)**

The arithmetic units can process four data items and output two data items per computation cycle. To support this high data processing rate, flexible interconnection among data processing units, data storage units, and I/O units are required. A general purpose crossbar (DX-Bar) serves this purpose. It has five input ports and seven output ports. The crossbar can be reconfigured every clock cycle under the control of microcode. The address crossbar (Ax-Bar) is used to select any one of the two address sources (i.e. external address bus and LAGU) and supplies addresses to the two data memories (M1 & M2).

• ***Input Queues*** **(XQ, YQ, adrQ)**

A fall through queue on each of the input channels increases the intercell bandwidth and also ensures that data and address are properly synchronized at the time of operation. Standard 512X8 FIFO chips (e.g. CY7C412) can be used for

this purpose.

• *Input multiplexer* (YMx) and XLth

These are used to support various modes of array configuration.

a.) *Forward mode* : In this mode, both X-data and Y-data items flow in the same direction i.e. from left to right.

b.) *Reverse mode* : In this mode , X and Y travels in opposite directions. While X flows from left to right, Y flows from right to left. This bi-directional data flow is required for many systolic algorithms (e.g. LU-Decomposition).

c.) *Wrap-around mode*: To handle large size problems, the algorithm has to be partitioned properly. In one partitioning method called LSGP scheduling (Chap-4), a single cell is multiplexed to perform the function of several cells. This requires output of a cell be fed back to the inputs which is supported in this mode. The XLth is used to hold X-data after it has been taken out from the XQ. This allows use of X-data a number of times once it has been transmitted to the cell.

• *Local address generation unit* (LAGU) *and microengine*.

This unit is the same as the one used in the interface cell and has already been discussed extensively in Chapter-5. The local address generator is used for data dependent addressing which is local to a particular cell.

§ 6.3 *DESIGN OF A SIMPLIFIED CELL*

Although the goal of this thesis was to design and implement a part of the interface unit for the systolic array, a small cell has been added to test the data transfer and data handling activities of the interface. For this

purpose, we have chosen three algorithms (matrix multiplication, linear convolution and AR filtering) for implementation, and a minimal cell structure was designed to execute these algorithms. The basic computational unit used here is a multiply-accumulator (MAC) chip; in addition, the cell has a local RAM of 8K capacity to store resident input data. Other inputs for computation come from either X or Y bus. For controlling the operation of the cell, a few bits of the microprogram residing on the interface have been used. All data paths of the cell are 8-bit wide and support only 8-bit operations.

§ 6.3.1 *BLOCK DIAGRAM DESCRIPTION OF THE CELL*

Block diagram of the cell is given in Fig. 6.3.

Following is a description of each block.

- ***Input latches*** **(XLth, YLth, O/PLth)**

These are used to latch the X and Y data coming from the interface and o/p data being returned to the interface. These latches introduce one stage of pipelining at each input and output.

- ***Buffers*** **(adrBf, DBf, O/PBf)**

Address and data are buffered wherever they are taken from or put onto the external bus. The DBf is used to buffer data on the global BC-bus and is used to load the local memory M1.

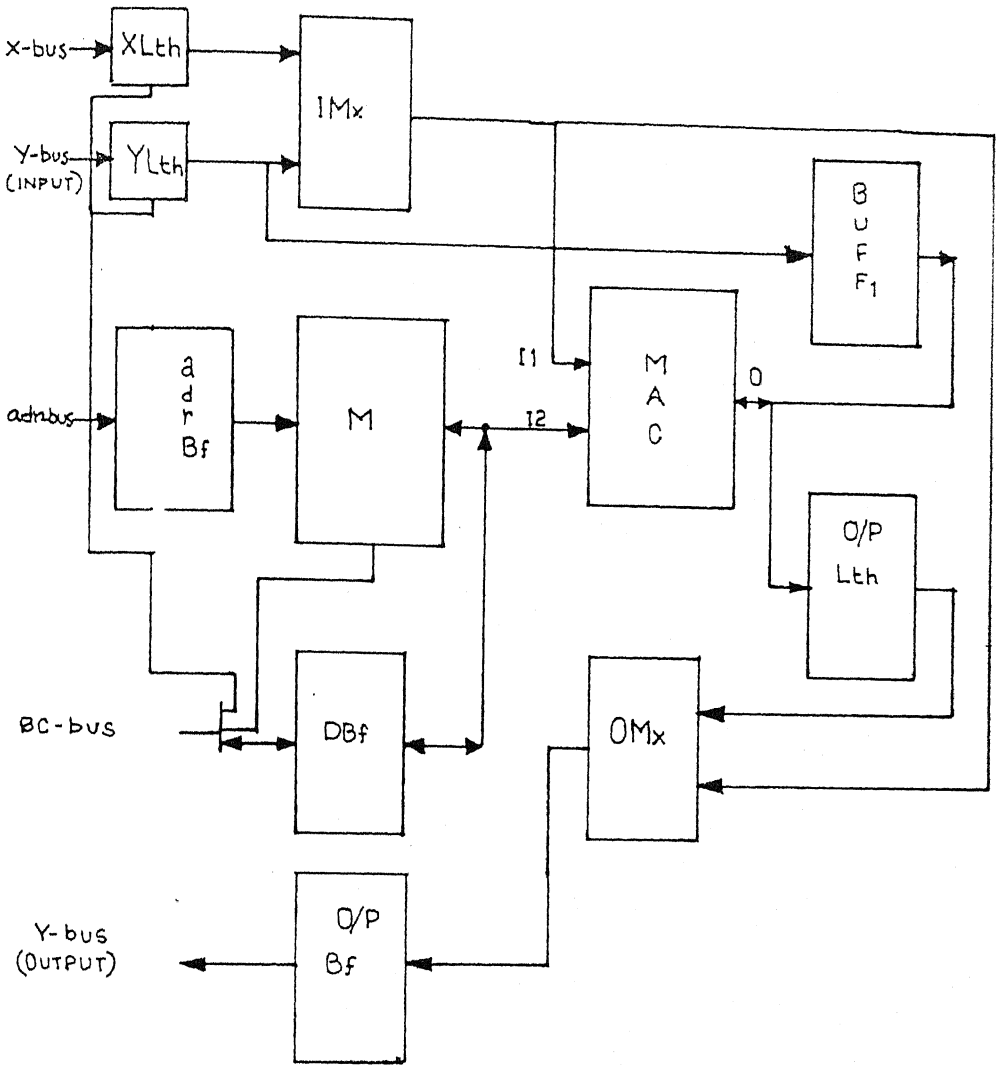


FIG. 6.3 Block diagram of the simplified cell

- ***Data multiplexers*** **(IMx, OMx)**

One input of the MAC can be chosen from either X or Y bus by selecting the proper IMx channel. The OMx gives a loop around facility for Y data input which can be recovered unchanged even if the MAC operates on the Y data. This feature is required in AR filtering problem where an output item is used to compute next N outputs and hence it has to be retained for those many cycles by looping it around.

• *Arithmetic unit* (MAC)

The multiply-accumulator does the following operation on its nth operation cycle:

$$O_{(n+1)} = O_n + I1_n + I2_n \ .$$

O_n can be the output of the operation done in the (n-1)th cycle, or it can be preloaded with Y data in the nth cycle itself. ADSP-1008A MAC chip has been used for arithmetic operation.

• *Data storage* (M)

An 8KX8 RAM has been used to store one set of inputs (e.g., system coefficients for AR filtering). The memory is preloaded by the interface prior to cell operation and is read during execution. In both cases it is addressed via the adr-bus by the address generator of the interface.

§ 6.3.2 *HARDWARE DESCRIPTION OF THE SIMPLIFIED CELL*

The schematic diagram of the cell is given in Fig. 6.4. The *-marked signals are bits from a microcode field of the interface . *Clk* is the system clock while *ClkB* is another clock having twice the frequency of *Clk*. The phase relation between *Clk* and *ClkB* is given in Fig. 6.6. Before starting execution, the local memory (M) has to be loaded with the data write facility of the interface. The address of the memory comes from the adr-bus which is the output of the interface AGU. To provide address through AGU, a small microprogram is first loaded and executed on the interface. The program initializes a register (R_i) of the 1410 and the 1410 outputs the contents of the register on the adr--bus. Simultaneously, the sequencer (1401) comes to a state where it waits for the *Flg*

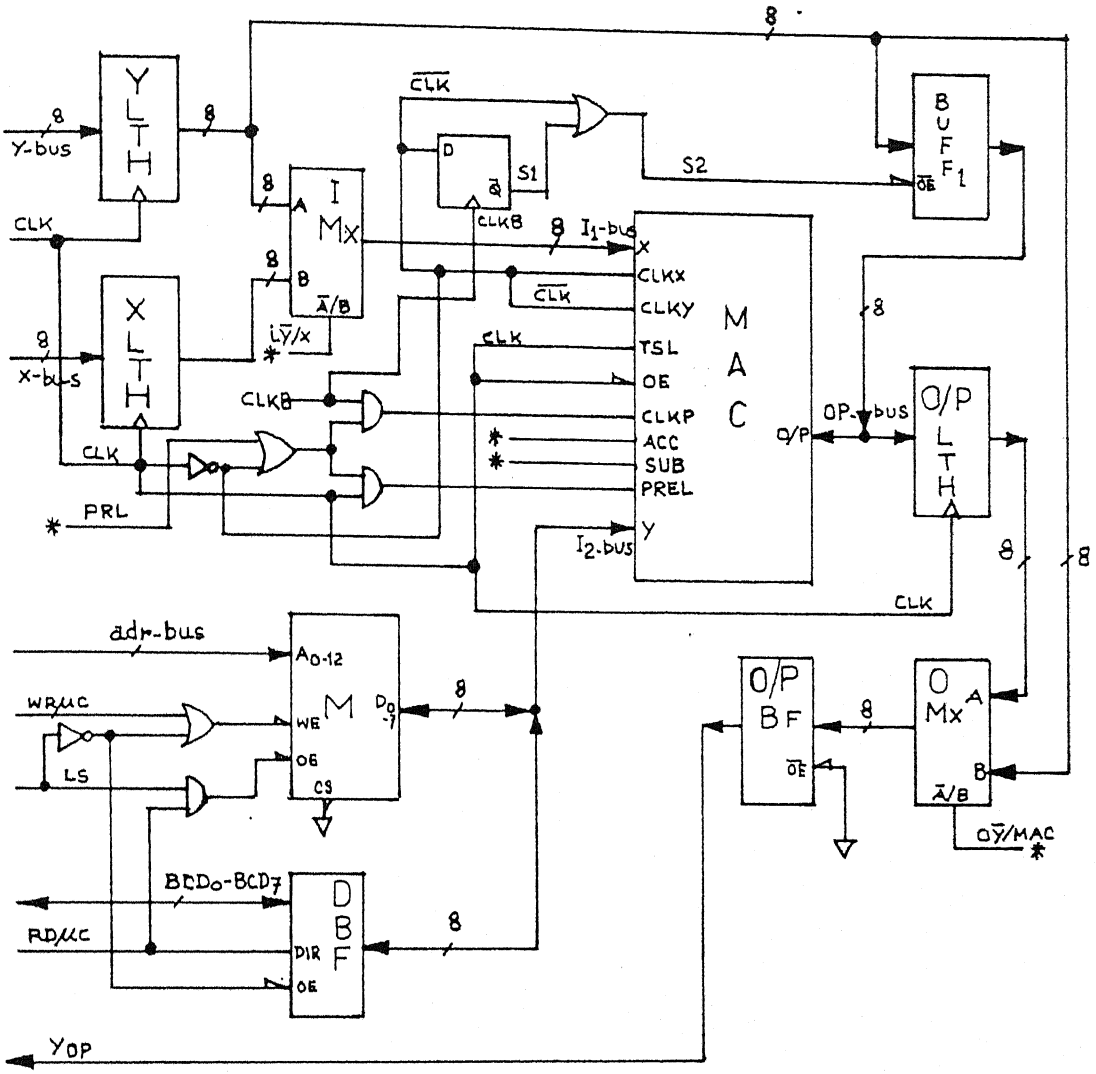


Fig. 6.4 Schematic Diagram of the Simplified Cell.

condition. Now, if the host has executed a data write into the cell (in this cycle a write operation is done on an I/O port called *mmc* with the control bits *lwt*=1 & *lμc*=0), data sent by the host will be written into 'M' at an address equal to the content of R_i. After the write has been finished, *Flg* to the sequencer is asserted and the sequencer moves over to the next microcode instruction where the 1410 modifies its R_i.The same process is repeated until the host finishes writing into 'M'. Flow chart of this process is shown in Fig. 6.5.

After finishing data loading in the manner described above, other data RAMs on the interface are written. Finally the microprogram, to execute the algorithm, is loaded and control is transferred to the array by making *Host/array* = *Lwt* = *Lμc* = 0. The MAC has two control lines (*Acc*, *Sub*) to specify the operation to be performed. These lines, alongwith the input data lines (I1, I2), are latched into the MAC at the second phase of the clock (*Clk*) and hence MAC operation is performed in the second phase only. The output of the operation is available at around the end of the clock cycle and it is latched into O/PLth at the rising edge of the next clock cycle during which the output is read by the interface. The *Acc* & *Sub* lines can be used to define the MAC operation as listed below.

	Acc	*Sub*	MAC operation
i)	1	1	$O_{(n+1)} = X_n * Y_n - O_n$
ii)	1	0	$O_{(n+1)} = X_n * Y_n + O_n$
iii)	0	x	$O_{(n+1)} = X_n * Y_n$

As mentioned before, O_n can be preloaded during the first phase of the nth cycle, while operation on O_n is performed in the second phase. During preloading, the *Prel* input of the MAC should be held high, output bus should be

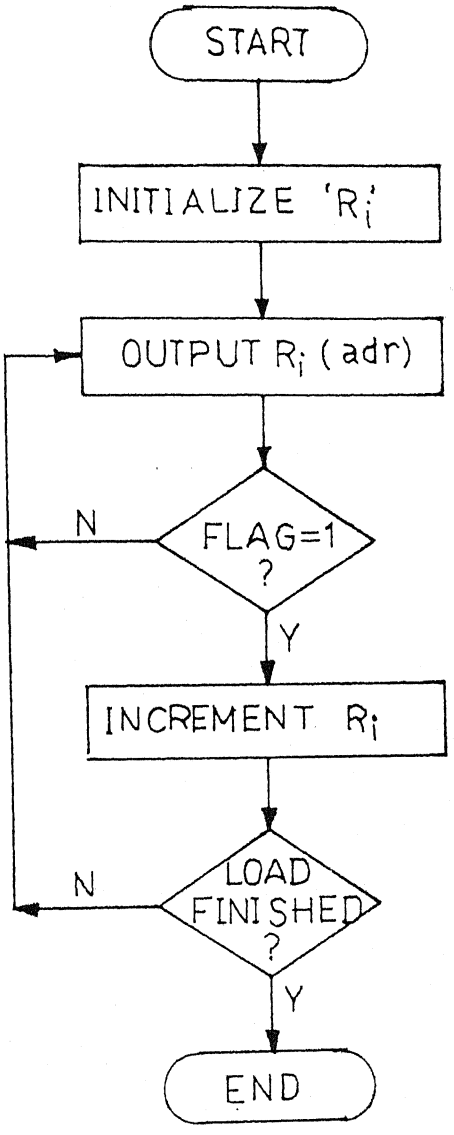


Fig. 6.5 Flow Chart for Address Generation During Data Loading.

tri-stated (*Tsl* = high) and under this condition data at the output port is preloaded at the rising edge of the *ClkP*. The data on the Y-bus can be preloaded through BUFFi in this design. For preloading microcode control bit *Prl* is made high. Other microcode bits are used to specify MAC operation and to select the IMx and O/PMx channels. Timing diagram of the operations are given in Fig. 6.6.

§ 6.4 *EXECUTION OF ALGORITHMS ON THE CELL*

The cell has been designed to execute three algorithms- matrix multiplication, linear convolution and AR filtering. The mapping methodology discussed in Chapter-3 can be used to map these algorithms on this single cell systolic array (!). But in case of a single cell structure the data dependencies of these algorithms are easily understandable and it is not really necessary to go through the trouble of projection, scheduling, scaling etc. The input and output sequence can be found directly through observation and are tabulated in Tables 6.1, 6.2, & 6.3.

It is to be noted that since a single cell has been used, the output is produced after sequential execution of a number of intermediate stages. Those intermediate results may have to be looped back into the cell through Y_B while the outputs are fed into Y_A.

With a single cell, obviously the advantages of parallel processing are not achieved but the merits of systolic data flow can still be felt. From the tables we see that the number of clock cycles required for execution is almost

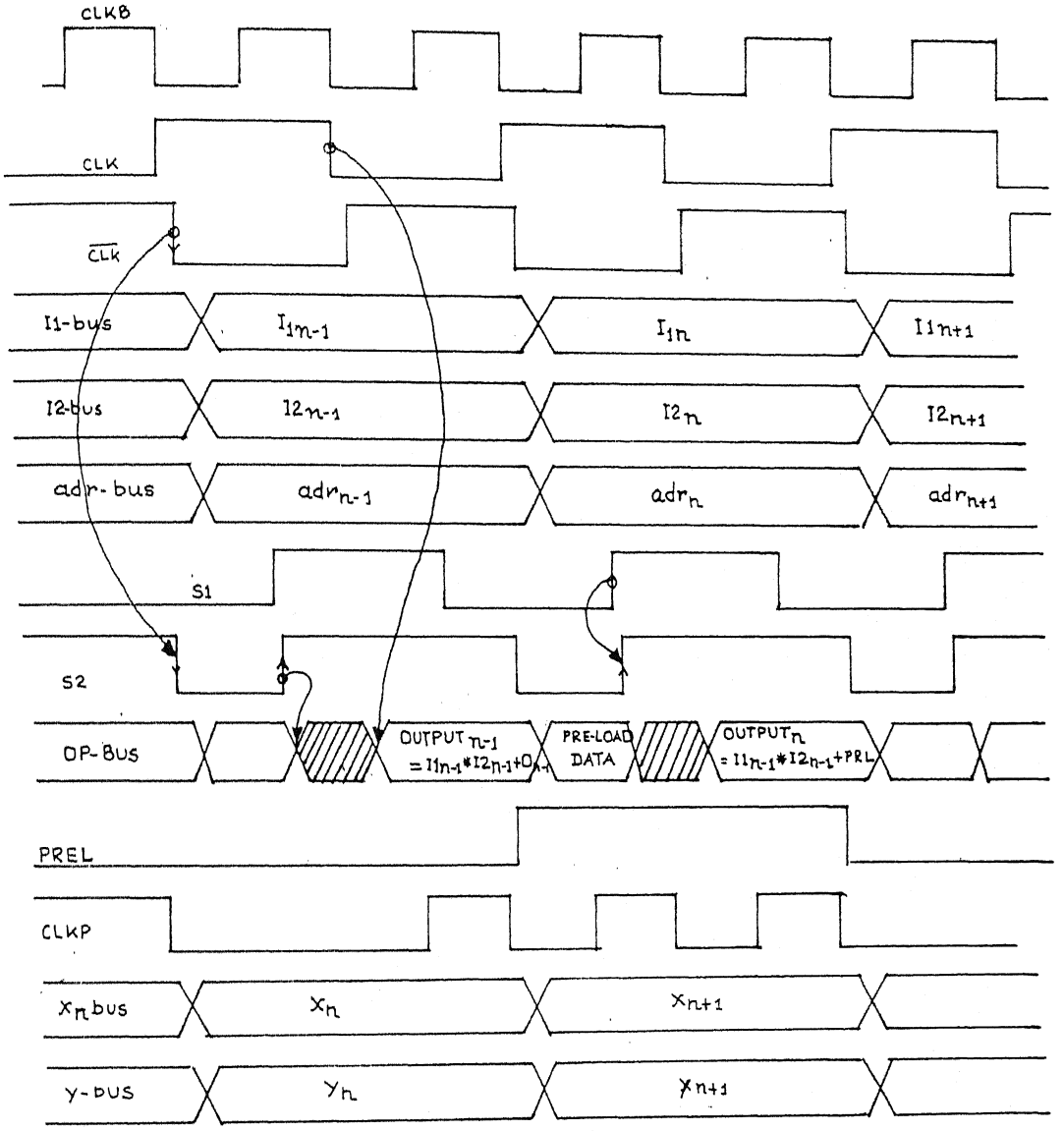


Fig. 6.6 Timing Diagram for Cell Operation.

equal to the number of arithmetic operations (multiply or multiply-add). I/O overload is almost negligible.

Microprograms for execution of the algorithms are given in appendix-2. The tight pipelining of I/O operation was not maintained in practice because of the following reasons.

The interface has been designed with the assumption that it should be connected to intelligent cells with their own control units. Accordingly a blocking scheme has been implemented, which blocks the clock of the appropriate cell in case of any overflow or underflow of data buffering units during intercell communication. Since the rest of the array continue with execution, normalcy is soon restored and the blocked cell is brought back into operation. But in present case, there is only one controller controlling the interface and the cell. If it goes into blocked state due to overflow or underflow of the Y_B RAM, there is no way of recovery from that state. Therefore, the operation cycles have been sparsed sufficiently to avoid data contention.

§ 6.5 *A SOFTWARE UTILITY TO USE 'SASP'*

To put SASP into service, the host has to initialize it with data and microprograms. These operations require frequent manipulation of control registers, communication with different I/O ports etc. To carry out these operations with simple commands, a program has been developed. This program enables the user to communicate with all the I/O ports in different access modes, e.g., single byte I/O, multiple I/O to transfer string of data, transfer of

files etc. These functions are executed at user commands followed by suitable command-tails depending upon the function. The details of the commands and functions performed by them have been listed in Appendix-3.

MATRIX MULTIPLICATION

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix} * \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \end{bmatrix} = \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ y_{31} & y_{32} & y_{33} \end{bmatrix}$$

CLOCK	INTERFACE OUTPUTS		MAC INPUTS			MAC OPERATION AND OUTPUT	INTERFACE INPUTS	
	X	Y	I1	PRE-LD.	I2		YA	YB
1.	a_{11}	-	-	-	-	-	-	-
2.	a_{12}	-	a_{11}	-	x_{11}	$y_{11}^1 = I1 * I2 = a_{11}x_{11}$	-	-
3.	a_{13}	-	a_{12}	-	x_{21}	$y_{11}^2 = a_{12}x_{21} + y_{11}^1$	-	-
4.	a_{14}	-	a_{13}	-	x_{31}	$y_{11}^3 = a_{13}x_{31} + y_{11}^2$	-	-
5.	a_{21}	-	a_{14}	-	x_{41}	$y_{11} = a_{14}x_{41} + y_{11}^3$	-	-
6.	a_{22}	-	a_{21}	-	x_{11}	$y_{21}^1 = a_{21}x_{11}$	y_{11}	-
7.	a_{23}	-	a_{22}	-	x_{21}	$y_{21}^2 = a_{22}x_{21} + y_{21}^1$	-	-
8.	a_{24}	-	a_{23}	-	x_{31}	$y_{21}^3 = a_{23}x_{31} + y_{21}^2$	-	-
9.	a_{31}	-	a_{24}	-	x_{41}	$y_{21} = a_{24}x_{41} + y_{21}^3$	-	-
10.	a_{32}	-	a_{31}	-	x_{11}	$y_{31}^1 = a_{31}x_{11}$	y_{21}	-
11.	a_{33}	-	a_{32}	-	x_{21}	$y_{31}^2 = a_{32}x_{21} + y_{31}^1$	-	-
12.	a_{34}	-	a_{33}	-	x_{31}	$y_{31}^3 = a_{33}x_{31} + y_{31}^2$	-	-
13.	a_{11}	-	a_{34}	-	x_{41}	$y_{31} = a_{34}x_{41} + y_{31}^3$	-	-
14.	a_{12}	-	a_{11}	-	x_{12}	$y_{12}^1 = a_{11}x_{12}$	y_{31}	-
15.	a_{13}	-	a_{12}	-	x_{22}	$y_{12}^2 = a_{12}x_{22} + y_{12}^1$	-	-
16.	a_{14}	-	a_{13}	-	x_{32}	$y_{12}^3 = a_{13}x_{32} + y_{12}^2$	-	-
17.	a_{21}	-	a_{14}	-	x_{42}	$y_{12} = a_{14}x_{42} + y_{12}^3$	-	-

TABLE. 6.1.

LINEAR CONVOLUTION

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} x_0 & 0 & 0 \\ x_1 & x_0 & 0 \\ x_2 & x_1 & x_0 \\ x_3 & x_2 & x_1 \\ 0 & x_3 & x_2 \\ 0 & 0 & x_3 \end{bmatrix} * \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix}$$

CLOCK	INTERFACE OUTPUTS		MAC INPUTS			MAC OPERATION AND OUTPUTS	INTERFACE INPUTS	
	X	Y	I1	PRE-LD.	I2		YA	YB
1.	x_0	0	—	—	—	—	—	—
2.	x_1	0	x_0	—	ω_0	$y_0 = I1 * I2 + PRL = x_0\omega_0$	—	—
3.	x_2	0	x_1	—	ω_0	$y_1^1 = x_1\omega_0$	y_0	—
4.	x_3	0	x_2	—	ω_0	$y_2^1 = x_2\omega_0$	—	y_1^1
5.	x_0	y_1^1	x_3	—	ω_0	$y_3^1 = x_3\omega_0$	—	y_2^1
6.	x_1	y_2^1	x_0	y_1^1	ω_1	$y_1 = x_0\omega_1 + y_1^1$	—	y_3^1
7.	x_2	y_3^1	x_1	y_2^1	ω_1	$y_2^2 = x_1\omega_1 + y_2^1$	y_1	—
8.	x_3	—	x_2	y_3^1	ω_1	$y_3^2 = x_2\omega_1 + y_3^1$	—	y_2^2
9.	x_0	y_2^2	x_3	—	ω_1	$y_4^1 = x_3\omega_1$	—	y_3^2
10.	x_1	y_3^2	x_0	y_2^2	ω_2	$y_2 = x_0\omega_2 + y_2^2$	—	y_4^1
11.	x_2	y_4^1	x_1	y_3^2	ω_2	$y_3 = x_1\omega_2 + y_3^2$	y_2	—
12.	x_3	—	x_2	y_4^1	ω_2	$y_4 = x_2\omega_2 + y_4^1$	y_3	—
13.		—	x_3	—	ω_2	$y_5 = x_3\omega_2$	y_4	—

TABLE 6.2.

AR-FILTERING

$$y_j = \sum_{K=1}^{N} a_K y(j-K) + u_j \qquad j = 1\ldots\ldots \quad \text{and,} \quad y_0 = y_{-1} = y_{-2} = y_{-3} = \ldots = 0$$

$$y_1 = u_1 + a_1 y_0 + a_2 y_{-1} + a_3 y_{-2} = u_1$$

$$y_2 = u_2 + a_1 y_1 + a_2 y_0 + a_3 y_{-1} = u_2 + a_1 y_1$$

$$y_3 = u_3 + a_1 y_2 + a_2 y_1 + a_3 y_0 = u_3 + a_1 y_2 + a_2 y_1$$

CLOCK	INTERFACE OUTPUTS		MAC INPUTS			MAC OPERATION AND OUTPUT	INTERFACE INPUTS	
	X	Y	I1	I2	PRE-LD		Y_A	Y_B
1.	u_1	0	—	a_3	u_1	—	—	—
2.	—	0	0	a_3	u_1	$y_1^1 = 0 \times a_3 + u_1$	—	—
3.	—	0	0	a_2	—	$y_1^2 = 0 \times a_2 + y_1^1$	—	0
4.	u_2	0	0	a_1	—	$y_1 = 0 \times a_1 + y_1^2$	—	0
5.	—	0	0	a_3	u_2	$y_2^1 = 0 \times a_3 + u_2$	y_1	y_1
6.	—	y_1	0	a_2	—	$y_2^2 = 0 \times a_2 + y_2^1$	—	0
7.	u_3	0	y_1	a_1	—	$y_2 = y_1 a_1 + y_2^2$	—	y_1
8.	—	y_1	0	a_3	u_3	$y_3^1 = 0 \times a_3 + u_3$	y_2	y_2
9.	—	y_2	y_1	a_2	—	$y_3^2 = y_1 \times a_2 + y_3^1$	—	y_1
10.	u_4	y_1	y_2	a_1	—	$y_3 = y_2 \times a_1 + y_3^2$	—	y_2
11.	—	y_2	y_1	a_3	u_4	$y_4^1 = y_1 \times a_3 + u_4$	y_3	y_3
12.	—	y_3	y_2	a_2	—	$y_4^2 = y_2 \times a_2 + y_4^1$	—	y_2
13.	u_5	y_2	y_3	a_1	—	$y_4 = y_3 \times a_1 + y_4^2$	—	y_3
14.	—	y_3	y_2	a_3	u_5	$y_5^1 = y_2 \times a_3 + u_5$	y_4	y_4

TABLE 6.3

Chapter 7

SUMMARY AND CONCLUSIONS

Systolic arrays combine the characteristics of array processing and pipelining in a regular pattern of interconnected nodes. Suitable algorithms are mapped on the array in such a way that the data dependencies of the computations match the interconnection pattern of the cells.

A systolic array signal processor (SASP) has been described. It is a linear array of microprogrammed cells connected to an external host through an interface cell which deals with all the data pumping and receiving operations. The host communicates with individual cells through the interface for loading microcode and data.

An interface cell has been designed and part of it has been implemented on hardware. A small cell has also been made and a few algorithms have been executed on this simple prototype. The interface can be accessed through a PC-XT and a program has been written for loading data and microcode onto the interface and for reading output from it.

To execute an algorithm efficiently on an array structure, it should be

mapped properly so that its data dependencies match the array interconnection pattern. Before settling on a design, a detailed study of the algorithms to be executed on the particular structure should be carried out to have a clear understanding of the different kinds of data routing that may be required during execution. This is to ensure that the final design supports a flexible data path which takes care of all the data dependencies. Although 'systolic description' of a handful of algorithms are readily available, different array structures are proposed for different types of algorithms. LU and QR decomposition algorithms, which are generally described on a mesh or hexagonal array have been mapped on linear array structure and this has helped to understand the demands on data passing mechanism. We feel that a more detailed and comprehensive study is required on the algorithm side for a more versatile design.

The SASP architecture has the attributes of a versatile computing machine. Because of time constraints we attempted to develop a simplified version using readily available simple LSI and MSI components which could not exploit all the features of the advanced IC's like 1401 and 1410 ; the flexibility of architecture has also not been fully implemented. Prime motivation of the current work was to have some experience with this architecture and also to check the feasibility and viability of such systolic array processor. To extract the full computing capability from a SASP type architecture we will require an enhancement of the current design with advanced high speed chips. A cycle time of 100ns can be achieved (this gives a throughput of 10N MFlops where N is the number of cells), whereas the present circuit with LS chips operate at a much reduced clock of 1 MHz only. Further increase in speed is possible with a

switchover to wavefront principle.

Software aids are required for programming the SASP. An optimizing compiler can hide the low-level details of the machine and allow the user to concentrate more on the parallelism at the array level. The W2 language devoloped for the CMU Warp processor [Annr87] is an example of such software support. But attempts in this direction can be made only after the system details are fully defined. This area has been beyond the scope of the current work and can be taken up as a future enhancement of the SASP system.

REFERENCES

ADSP '1987 DSP Products Data Book', Analog Devices Inc. 1987.

Alg85 'Algorithmically Specialized Parallel Computers', Ed. by, Snyder L., et al, Academic Press, INC., 1985.

Annr87 Annaratone Macro, et al, 'The Warp Computer: Architecture, Implementation, and Performance', IEEE Trans. on Computers, Vol.C-36, No. 12, Dec 1987.

Arn185 Arnould E., et al, 'A Systolic Array Computer', Proc. of IEEE Int'l Conference on ASSP-ICASSP85.

Bat85 Batcher K.E., ' MPP : A High Speed Image Processor', 'Algorithmically Specialized Parallel Computers', Ed. by, Snyder L., et al, Academic Press, INC., 1985.

Burt84 Burt P.J., ' The Pyramid as a Structure for efficient computation', 'Multiresolution Image Processing and Analysis', Ed. by, A. Rosenfield, Springer-Verlag 1984.

Dew84 Dew P.M. 'VLSI Architectures for Problems in Numerical Computation', Super Computers and Parallel processing, Oxford Science publication 1984.

Dy82 Dyer C.R., 'Pyramid Algorithms and Machines', Multicomputers and Image Processing, Ed. by K. Pestone, Academic Press 1982.

HTKung82 Kung H.T., ' Why Systolic Architectures ', Computer Magazine, Jan. 1982.

HTKung84 Kung H.T., ' Systolic Algorithms on CMU WARP Processor', Proc. of Int'l Conference on Pattern Recognition, Vol.1, 1984.

Leis83 Leiserson C.E., 'Area Efficient VLSI Computation', The MIT Press 1983.

Laz86 Lazou C., 'Supercomputers and their use', Clarendon Press, Oxford 1986.

Moldo83 Moldovan D.I., 'On the Design of Algorithms for VLSI Systolic Arrays', Proceedings of the IEEE, Vol. 71, No. 1, Jan 1983.

Moldo86 Moldovan D.I., et al, 'Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays', IEEE Transactions on Computers, Vol. C-35, No. 1, Jan 1986.

Navar87 Navarro J.J, et al, 'Partitioning: An essential Step in Mapping Algorithms Into Systolic Array Processors', Computer Magazine, pp. 77-89, July 1987.

Nemm88 Nemawarker S.S. ' SASP : A Systolic Array Signal Processor', M. Tech Thesis, IIT-Kanpur, May 1988.

Quin86 Quinton P., 'Introduction to Systolic Architectures', 'Lecture Notes on Computer Science, Vol. 272, 'Future Parallel Computers', Springer Verlag, 1986.

SYKung85 Kung S.Y., et al, 'Eigenvalue, Singular Value and Least Square Solvers via the Wavefront Array Processor', 'Algorithmically Specialized Parallel Computers', Ed. by, Snyder L., et al, Academic Press, INC., 1985.

SYKung87 Kung S.Y., et al, 'Wavefront Array Processors—Concept to Implementation', IEEE Computer Magazine, pp. 18-33, July 1987.

SYKung88 Kung S.Y., 'VLSI array processors', Prentice Hall, 1988.

Stone85 Stone H.S., 'Special Purpose vs.General Purpose Systems: A Position Paper',

Uhr84 Uhr L., 'The several steps from Icon to Symbol Using Structured Cone', 'Multiresolution Image Processing and Analysis', Ed. by, A. Rosenfield, Springer-Verlag 1984.

Uhr85 Uhr L. 'Pyramid Multicomputers, and Extensions and Segmentations', 'Algorithmically Specialized Parallel Computers', Ed. by, Snyder L., et al, Academic Press, INC., 1985.

Us89 Usman Mohd., 'Design of SASP: A Systolic Array Signal Processor', M.Tech Thesis, IIT-Kanpur 1989.

Overview of ADSP 1401

ADSP-1410 OVERVIEW

Digital Signal Processing (DSP) and array processing systems require fast, flexible address generation circuitry. An Address Generator (AG) supplies the address of a location in data or coefficient memory. The value residing at the specified address is fetched and fed to an arithmetic unit for processing. The AG must then modify the address pointer in anticipation of the next data fetch. For algorithms that repetitively loop through data buffers, the AG may need to compare the address to a buffer end and conditionally loop back to the top of the buffer. Finally, to maximize throughput, an AG must perform its addressing tasks rapidly and without overhead.

With the ADSP-1410, 16-bit pointers to memory are stored in an address (R) register file. Since an AG must track several pointers concurrently, sixteen R registers, denoted R_n, are provided. If we denote Y as the address port, the operation "$Y \leftarrow R_n$" corresponds to the AG supplying an address from register R_n.

After supplying an address, the AG must update the pointer for the next memory fetch. The updating may be as simple as an increment but, more generally, involves adding or subtracting an arbitrary offset value. Also, algorithms generally access several different offset values. To this end, the AG provides six offset registers, denoted B_m, and can execute in a single-cycle the core operation:

$$Y \leftarrow R_n;\ R_n \leftarrow R_n + B_m.$$

In DSP applications, data arrays are often addressed as circular buffers. That is, when addressing reaches the buffer end, it wraps back to the beginning of the buffer. To implement this looping, the AG compares the supplied address to one of four compare registers, denoted C_j. If the address has moved to or beyond the end of the boundary ($R_n \geq C_j$), the device can transfer an initialization register value, denoted I_j, to the register ($R_n \leftarrow I_j$); otherwise, it is updated in normal fashion ($R_n \leftarrow R_n + B_m$). To minimize overhead, the AG can execute normal updates while also performing conditional re-initializations; again, in one core operation:

$$Y \leftarrow R_n;\ \text{IF } (R_n \geq C_j):\ R_n \leftarrow I_j;\ \text{ELSE } R_n \leftarrow R_n + B_m.$$

Since the above instruction handles the looping required of circular buffer addressing, it is termed a looping instruction. To a large extent, the ADSP-1410's architecture and instruction set revolve around efficient implementation of this instruction. However, many variations of this instruction are supported on the device and spelled out in the following sections.

ADDRESS SOURCES

- Sixteen internal R registers
- External data provided over the D port

OFFSET SOURCES

- Six internal B registers
- Data Port

OFFSET OPERATIONS

- Increment ($R_n \leftarrow R_n + 1$)
- Decrement ($R_n \leftarrow R_n - 1$)
- Add Offset ($R_n \leftarrow R_n + B_m$)
- Subtract Offset ($R_n \leftarrow R_n - B_m$)
- Single-Bit Left/Right Shifts
- Logical Operations (AND, OR, XOR)

CONDITIONAL RE-INITIALIZATION

- Independent Inhibit/Enable for each of four initialization registers
- Conditional AIR execution (used for true modulo addressing)

OUTPUT-UPDATE SEQUENCE

- Normal (Pre-Update) Mode (output the address before update)
- Post-Update Mode (output the address after update)

PRECISION

- Single chip supplies 16-bit addresses
- Two chips cascaded provide 30-bit addresses
- One chip provides 30-bit addresses in two cycles

ADSP-1410 PIN ASSIGNMENTS

PIN NAME	DESCRIPTION
$Y_{15} - Y_0$	The address (Y) output port. In single-chip/double-precision mode, the MSB (Y_{15}) indicates whether the supplied address is the MSW or LSW (see Precision Modes). In two-chip/double-precision mode, the MSB conveys the carry/shift bit from the Least Significant (LS) to the Most Significant (MS) chip.
$D_{15} - D_0$	The bi-directional data (D) port. In two-chip/double-precision addressing mode, the MSB (D_{15}) of this port conveys CMP status from the partner chip.
$I_9 - I_0$	The instruction port.
CMP/Z	A dual function pin. Looping instructions, which compare address register values to compare register values, assert this pin HI to convey CMP status if i) $R \geq C$ for positive offsets, or ii) $R \leq C$ for negative offsets. Logical/Shift instructions assert this pin HI to convey the ZERO status of the result.
DSEL	Data Select control. Asserting this control HI causes data set up on the data port to substitute for the R value specified in the instruction.
AIR Enable	Alternate Instruction Register control. Asserting this control HI causes the device to execute an instruction stored in the internal AIR, rather than the instruction set up on the instruction port.
CLK	Clock
V_{dd}	+5 Volt Power Supply
GND	Ground

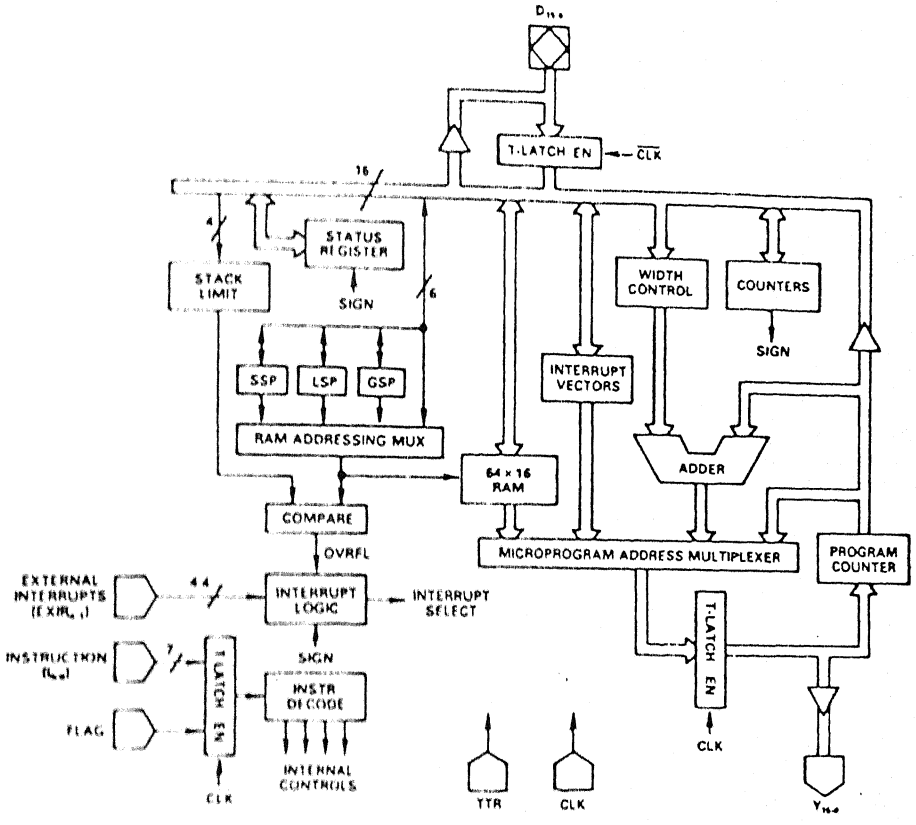


Figure 1. ADSP-1401 Block Diagram

ADDRESSING MODES

Direct: both absolute and relative
Indirect: from internal RAM

HARDWARE FEATURES

Instruction Port
Bidirectional Data Port
Four Input Address Multiplexer
Three Stack Pointers
Four Event Counters
Condition Flag
Eight Prioritized and Maskable User Interrupts
TTR Pin:
- Trap
- Three-State
- Reset

INSTRUCTION TYPES

Jumps and Branches
Stack Operations
Status Register Operations
Counter Operations
Interrupt Control
Relative Address Width Controls
Instruction Hold Control
Writeable Control Store
Dedicated Counter Underflow Interrupt
Dedicated Stack Overflow Interrupt

ADSP-1401 PIN ASSIGNMENTS

Pin Name	Description
$I_6 - I_0$	The 7-bit microinstruction controlling the ADSP-1401.
$Y_{15} - Y_0$	Output bus which provides addresses to the microprogram memory.
$D_{15} - D_0$	Bidirectional Data bus for transferring data to or from the ADSP-1401.
$EXIR_{4-1}$	Four external interrupt request lines. Note that internal circuitry supports 8 interrupts with the aid of an external 2 to 1 multiplexer.
CLK	External clock input
FLAG	An input used for conditional instructions. Its source is usually a condition multiplexer.
TTR	A multi-purpose pin accommodating traps, output disable and reset.
V_{DD}	+5 Volt supply.
GND	Ground.

fall time and capacitance measurement, we can determine that the peak current in each driver is:

$$I_{peak} = C_{load} \Delta V/\Delta t,$$

where $\Delta V/\Delta t$ is the initial slew rate.

In the case of the program sequencer, for an external load capacitance of 50pF and a measured slew rate of 0.6V/ns, the peak current will be about 30mA. Since there are 16 such drivers, the total peak current may approach 480mA!

The internal ground and supply lines may undergo a large disturbance during this transition unless the ADSP-1401 is tied to a solid ground plane and good high frequency decoupling is used (0.1μF ceramic between GND and V_{DD} as close as possible to the device). Otherwise, is it possible that internal data in the ADSP-1401 may be lost.

4.5 Mnemonics and Opcodes

Opcode bits "ii" select the relevant register (R_{3-0}) and/or counter (C_{3-0}). Opcode bits "cc" select the condition to be applied:

'00' UNCONDITIONAL
'01' NOT FLAG
'10' FLAG
'11' SIGN

The SIGN condition is precluded from instructions prefixed with "*".

Mnemonic	Opcode (I_{8-0})	Description
Jump and Branch Instructions:		
JFCOF	001 0101	IF FLAG: JUMP PC (*self*)
JFCNF	011 0101	IF NOT FLAG: JUMP PC (*self*)
JTWO	101 cc01	IF COND: JUMP PC + 2 (*skip*)
JDA	111 cc11	IF COND: JUMP DATA, ABSOLUTE
JDR	111 cc01	IF COND: JUMP DATA, RELATIVE
JDI	101 cc10	IF COND: JUMP DATA, INDIRECT
JDRST	100 11ii	IF SIGN OF C_i: JUMP DATA, C_i<- R_i, ELSE, C_i<- C_i - 1
*JRC	110 ccii	IF COND: JUMP R_i
JRS	110 11ii	IF SIGN OF C_i: JUMP R_i, C_i<- C_i - 1
JSA	111 cc00	IF COND: JUMP SUB, ABSOLUTE
JSR	111 cc10	IF COND: JUMP SUB, RELATIVE
RTN	101 cc11	IF COND: RETURN FROM SUB
*BRANCH	100 ccii	IF SIGN OF C_i: JUMP R_i; ELSE, C_i<- C_i - 1, IF COND: JUMP DATA
Stack Operations:		
Submume Stack		
PSDSS	001 1110	PUSH DATA ONTO SS
PPSSD	011 1110	POP SS TO DATA PORT
WRSSP	000 1110	WRITE SSP
RDSSP	010 1100	READ SSP
DSSP	000 0010	DECREMENT SSP
Register Stack		
SGSP	000 0111	SELECT GSP
SLSP	000 0110	SELECT LSP
RDRSP	010 1111	READ RSP
WRRSP	000 1100	WRITE RSP
PSPC	010 0011	PUSH PC ONTO RS
PSGSP	000 0101	PUSH GSP ONTO SS
PPGSP	000 0100	POP GSP FROM SS
PSDRS	001 1111	PUSH DATA ONTO RS
PPRSD	011 1111	POP RS TO DATA PORT
AIRSP	010 10ii	ADD i TO RSP
S1RSP	000 1111	SUBTRACT 1 FROM RSP
S4RSP	011 1100	SUBTRACT 4 FROM RSP

Status Register Bit Assignments

Bit#	Function (HI/LO)
SR_{15}	IR_9 Mask Bit
.	.
.	.
.	.
SR_4	IR_0 Mask Bit
SR_{3-4}	Relative Jump Width Selection: '00' = 16-bit relative address width '01' = 8-bit width '10' = IHC Mode (8-bit width) '11' = 12-bit width
SR_3	Select GSP/LSP
SR_2	Enable/Disable Interrupts
SR_1	Set/Clear Sign Bit
SR_0	Select Transparent/Latched Interrupts

Status Register Operations:		
RDSR	010 1110	READ SR
WRSR	001 1100	WRITE SR
PSSR	010 0001	PUSH SR ONTO SS
PPSR	010 0010	POP SR FROM SS
Counter Operations:		
WRCNTR	011 10ii	WRITE C_i
CLRS	001 0100	CLEAR SIGN BIT
SETS	011 0100	SET SIGN BIT
PSCNTR	000 10ii	PUSH C_i ONTO SS
PPCNTR	001 10ii	POP C_i FROM SS
DCCNTR	011 00ii	DECREMENT C_i
IFCDEC	101 cc00	IF COND: DECREMENT C_0
Interrupt Control:		
CCIR	001 0001	CLEAR CURRENT INTERRUPT
CAIR	000 0001	CLEAR ALL INTERRUPTS
RTNIR	000 0011	RETURN FROM INTERRUPT
RDIV	010 1101	READ INTERRUPT VECTOR AND INCREMENT IVP
WRIV	000 1101	WRITE INTERRUPT VECTOR AND INCREMENT IVP
IRMBC	001 0011	IR MASK BITWISE CLEAR
IRMBS	001 0010	IR MASK BITWISE SET
DISIR	001 0110	DISABLE INTERRUPTS
ENAIR	011 0110	ENABLE INTERRUPTS
SLIR	001 0111	SELECT LATCHED INTERRUPTS
STIR	011 0111	SELECT TRANSPARENT INTERRUPTS
SLRIVP	001 1101	WRITE SLR_{5-0} = D_{5-0} AND IVP_{5-0} = D_{15-12}
Relative Address Width Controls:		
REL16	010 0100	SELECT 16-BIT RELATIVE ADDRESSING
REL12	010 0111	SELECT 12-BIT RELATIVE ADDRESSING
REL8	010 0110	SELECT 8-BIT RELATIVE ADDRESSING
Miscellaneous Instructions:		
CONT	000 0000	CONTINUE
IDLE	001 0000	IDLE
IHC	010 0101	ENABLE INSTRUCTION HOLD CONTROL
WCS	010 0000	WRITE CONTROL STORE

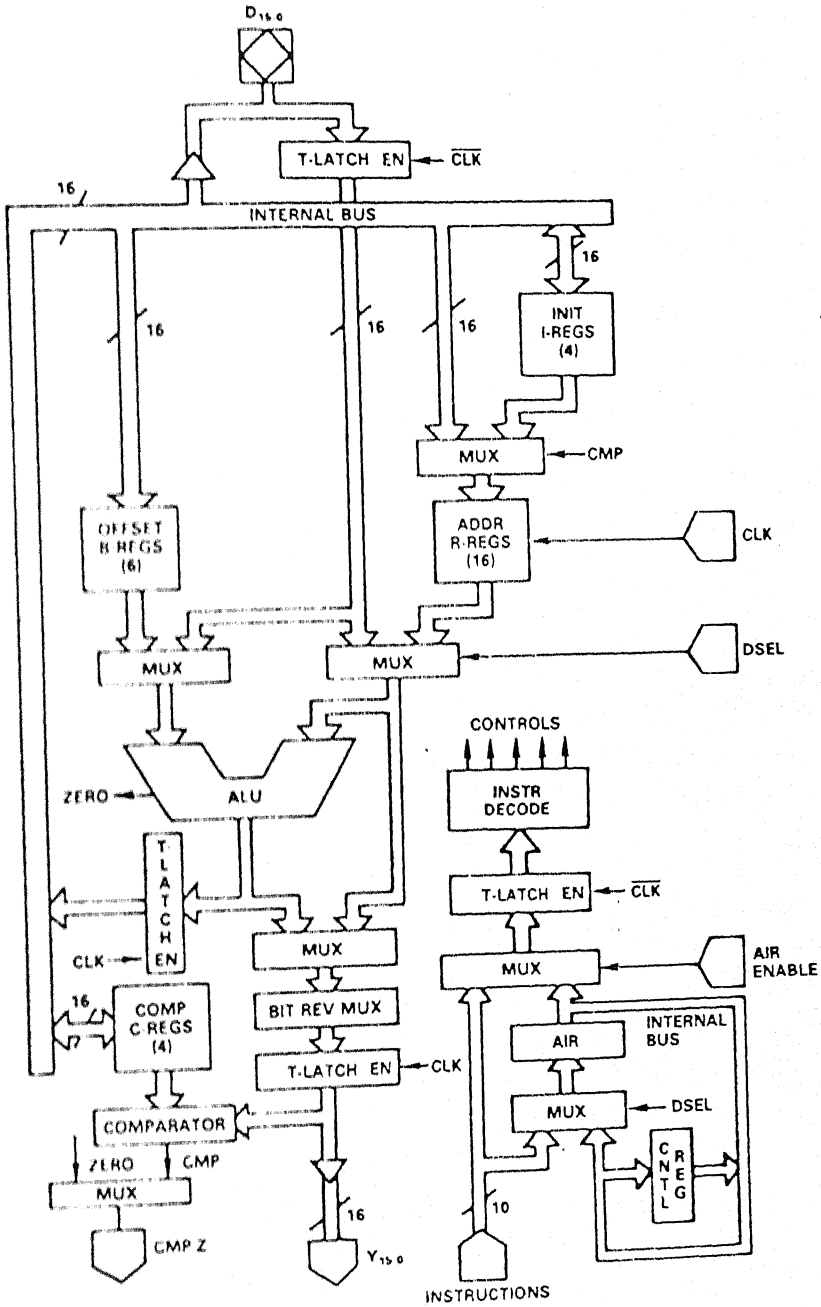


Figure 1. ADSP-1410 Functional Block Diagram

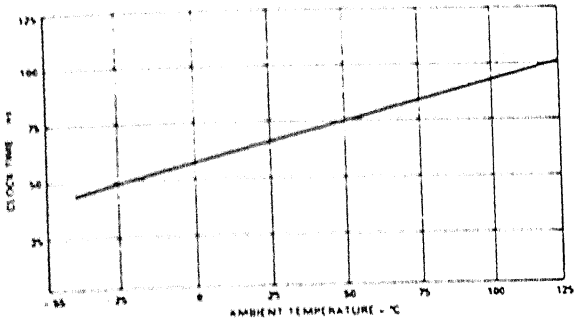


Figure 10. Clock Cycle Time vs. Temperature

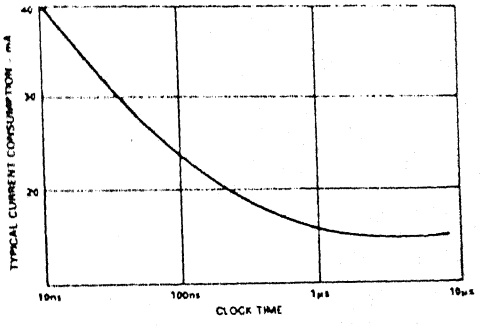


Figure 11. Typical I_{DD} vs. Frequency of Operation

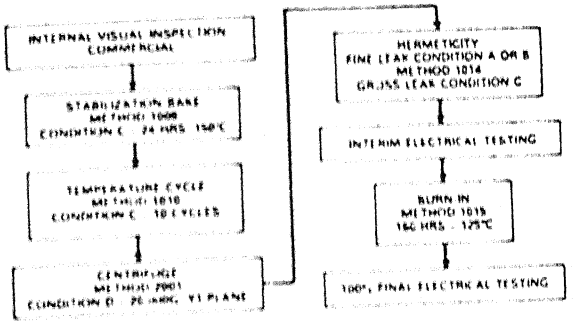


Figure 12. PLUS Processing Environmental Flow

MNEMONICS AND OPCODES

The following list gives the instruction mnemonics and opcodes. Various parameters are substituted by the user, defining register numbers or control bits. The notation convention is this:

R	=	Address register
B	=	Base (offset) register
C	=	Compare register
I	=	Initialization register
D	=	Data bus
CR	=	Control register
rrrr	=	Four-bit address register number
rrr	=	Three-bit address register number
bb	=	Two bit base (offset) register number
cc	=	Two bit comparison register number
ii	=	Two bit initialization register number
pp	=	Two bit precision code
x	=	One bit control bit

*External data may substitute for R using DSEL.
†Operable in either pre- or post-update mode.

Instr.	Opcode (I_{9-0})	Description
Looping Instructions		
YINC*†:	1011ccrrrr	output & increment/init
YDEC*†:	1010ccrrrr	output & decrement/init
YADD*†:	11ccbb1rrr	output & add offset/init
YSUB*†:	11ccbb0rrr	output & subtract offset/init
Register Transfer Instructions		
YRTR*:	000101rrrr	output & xfr R to R
YRTB*:	0011bbrrrr	output & xfr R to B
YRTC*:	0010ccrrrr	output & xfr R to C
DTI:	000011111i i	xfr D to I
ITR:	1000i irrrr	xfr I to R
BTR:	0100bbrrrr	xfr B to R
RTD:	000100rrrr	xfr R to D
CTD:	00001100cc	xfr C to D
BTD:	00001101bb	xfr B to D
ITD:	00001110i i	xfr I to D
Logical and Shift Instructions		
YOR*†:	0111bbrrrr	output & OR B with/to R
YAND*†:	0110bbrrrr	output & AND B with/to R
YXOR*†:	0101bbrrrr	output & XOR B with/to R
YASR*†:	000111rrrr	output & arith SR R to R
YLSL*†:	000110rrrr	output & logical SL R to R
Control Register Instructions		
RST:	0000000001	reset CR
DTCR:	0000101110	xfr D to CR
CRTD:	0000101111	xfr CR to D
SETI:	0000100i i x	set cond re-init on CMP mode
SETP:	00001010pp	set chip precision
SETY:	000001001x	set Y port to trans/latched mode
SELR:	000001101x	select upper/lower R bank
SELB:	000001100x	select upper/lower B bank
SETU:	000001011x	set post/pre update mode
SETA:	000001010x	set cond AIR mode
AIR Instructions		
WRA:	0000101100	write AIR with D
RDA:	0000101101	read AIR at D
LDA:	0000011110	load AIR on next cycle
Misc. Instructions		
YDTY:	0000011111	pass D to Y port
YREV*†:	1001bbrrrr	output R in bit-reverse format
NOP:	0000000000	no operation

MICROCODE FOR MATRIX MULTIPLICATION

```
00 00 00 00 72 70 00 60 6a 00 fe 62 00 fe 6a
ff 05 05 00 00 00 ff ff ff 0b 06 ff 01 05 ff
ff 00 00 00 80 80 ff ff ff 00 00 ff 01 00 ff
00 20 34 30 40 40 40 b0 b0 b0 00 30 c6 20 00
40 20 20 20 00 00 01 01 01 00 00 00 00 00 00
78 88 88 88 88 00 80 8a 8a 8a 88 88 88 88 88
f0 f0 f0 f0 f0 f0 f0 f0 f2 f2 f8 f0 f0 f0 f0
```

MICROCODE FOR CONVOLUTION

```
00 00 72 70 00 00 60 fe 00 62 fe 70 00 00 60 fe 00 00 00 6a
ff 00 00 00 ff ff ff 07 ff ff 04 00 ff ff ff 0f ff ff ff ff
ff 00 00 00 ff ff ff 00 ff ff 00 00 ff ff ff 00 ff ff ff ff
20 14 14 14 14 14 14 14 14 14 b0 14 14 14 14 14 14 14 14 14
20 20 00 05 05 05 0d 0d 09 08 08 05 05 05 05 05 01 00 00 00
88 88 88 88 80 80 88 80 80 88 88 80 80 80 88 80 80 80 80 80
f0 f0 f0 f0 f3 fb f3 f3 f3 f0 f0 f0 f3 fb fb fb fb f8 f8 f0
```

MICROCODE FOR AR-FILTERING

```
00 00 00 00 00 70 72 00 00 72 62 ff 60 ff 6a e0
ff 40 41 ff 00 08 00 ff ff 02 ff 0a ff 07 ff 07
ff 00 00 00 00 80 80 ff ff 80 ff 00 ff 00 ff 00
00 14 14 28 30 00 00 40 14 b8 14 b8 00 40 00 00
40 20 60 a0 a0 00 00 81 44 80 04 88 08 80 00 00
58 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88
f0 f0 f0 f0 f0 f0 f0 f0 f0 e2 f2 c2 f8 f0 f0 f0
```

B. row i represents the code contents of RAMi of the microcode M-bank.

Appendix – 3

Manual for using SASP

File Name SASP

Execute SASP to get the prompt SASP>. Services provided under this prompt are listed below. Some of the commands are followed by command-tails containing a number of arguments listed inside the brackets and separated by commas. To get a particular service, the corresponding command should be given with proper arguments. Command and individual arguments should be separated from each other by a blank.

1. O <*port name, data*(optional)>

   Outputs one byte ***data*** to the ***port***. In case no new data is given, the previous data is output.

2. I <*port name*>

   Inputs one byte from ***port*** and displays it on the screen.

3. LO <*port name, data, number*>

   Outputs ***data*** to the ***port*** continuously for a number of iterations specified in the ***number*** field of the command-tail. An infinite loop can be implemented by giving only the first two arguments and omitting the ***number***.

4. LI *<port name, number>*

   Inputs 8-byte data from the ***port*** continuously for a number of iterations specified by the ***number***. Infinite loops can be made by omitting the ***number*** argument.

5. FLOAD *<port name, full path description of the file>*

   Loads the file specified in the ***path*** into the ***port***.

6. CLCTR

   Clears all the counters which generate local addresses for the data RAMs.

7. WCS

   Gives a ***Wcs*** instruction to the sequencer.

8. SWM

   Switches from one microcode RAM to the other by giving a ***Wcs*** instruction to the sequencer (see Chap-5).

9. START

   Signals the sequencer to start execution, by setting the control and status registers.

LIST OF PORTS

R0, R1, R2 →	Control registers.	MMC →	Microcode RAM.
MX →	X-RAM.	MWT →	Data RAM of the cell.
MYA, MYB1 →	Y-RAMs.		